

VISUAL TIME-SAVING REFERENCE

SwiftUI Views



Quick Start



Version: 24-APRIL-2023

©2023 Big Mountain Studio LLC - All Rights Reserved

Special Thanks from Big Mountain Studio

As a new member of Big Mountain Studio you now get **10% off** all of our SwiftUI books!



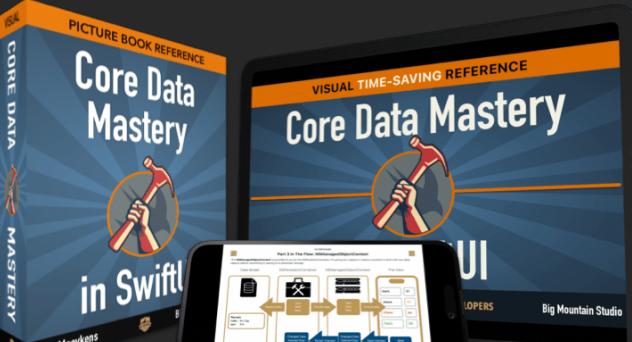
\$34 **\$30.60**



\$55 **\$49.50**



\$55 **\$49.50**



\$97 **\$87.30**



\$233 **\$209.70**

[CLICK HERE TO GET 10% OFF](#)

Book Quick Links



Foreword by Chris Ching



I've been teaching iOS app development to non-coders and beginners since 2013 and without fail, Apple changes things up to improve the developer experience every year.

As an iOS educator, it used to frustrate me because I'd have to update all of my training and content to reflect the new changes.

However, in retrospect, it was this annual song and dance that forced me to improve in my craft. It forced me to reflect on where my students were getting confused or stuck. It forced me to make my training better.

Reading Mark Moeykens' books gives me that same feeling.

I'm inspired by his ability to **break down complex concepts into relatable analogies** and his use of simple imagery to convey what a page of words can't.

That's why I always get new ideas about how I can improve my own training whenever I read his books.

When it comes down to it, a learning resource is only as valuable as its ability to transfer knowledge to you, the reader.

And for that reason, I think you have something of incredible value in your hands.

I've learned a lot from Mark and I think you will too.

Chris Ching

Chris Ching
codewithchris.com

HOW TO USE

This is a visual **REFERENCE GUIDE**. Find a screenshot of something you want to learn more about or produce in your app and then read it and look at the code.



Read what is on the screenshots to learn more about the views and what they can do.

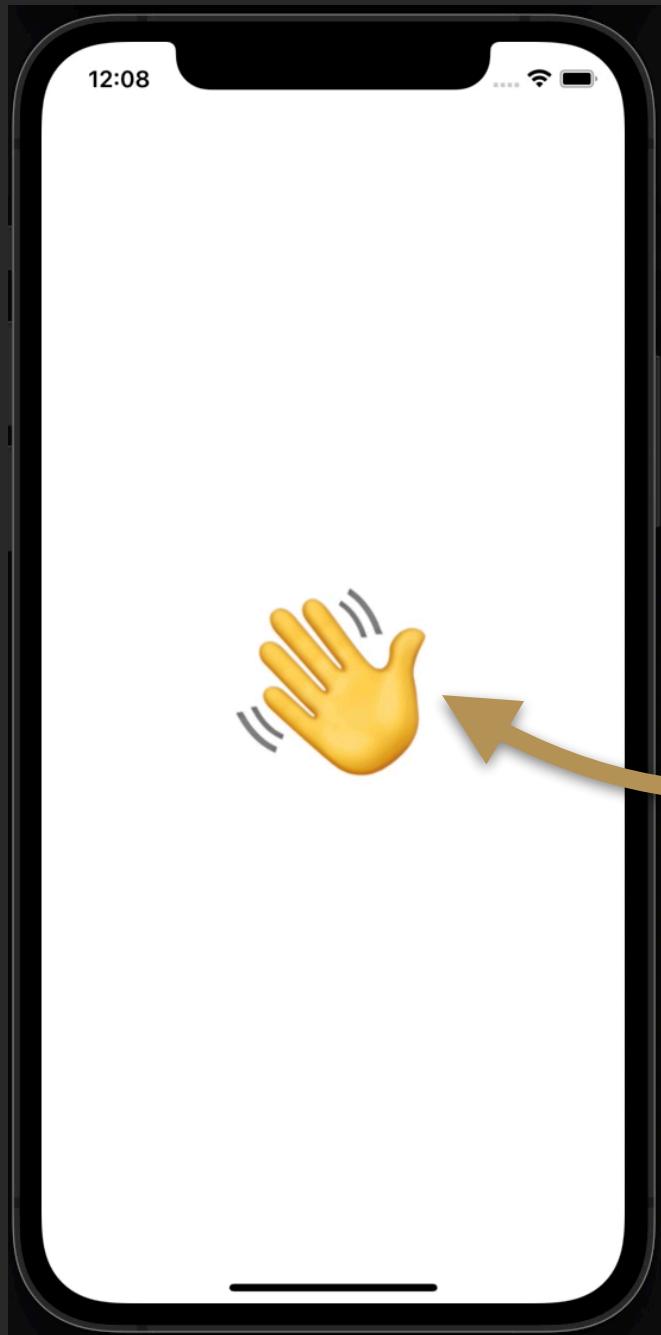
You can also read the book from beginning to end. The choice is yours.



CONVENTIONS



Embedded Videos



The ePUB version of the book supports embedded videos.

The PDF version does not.

This play button indicates that this is a playable video in the ePUB format.

But in PDF it renders as simply a screenshot.

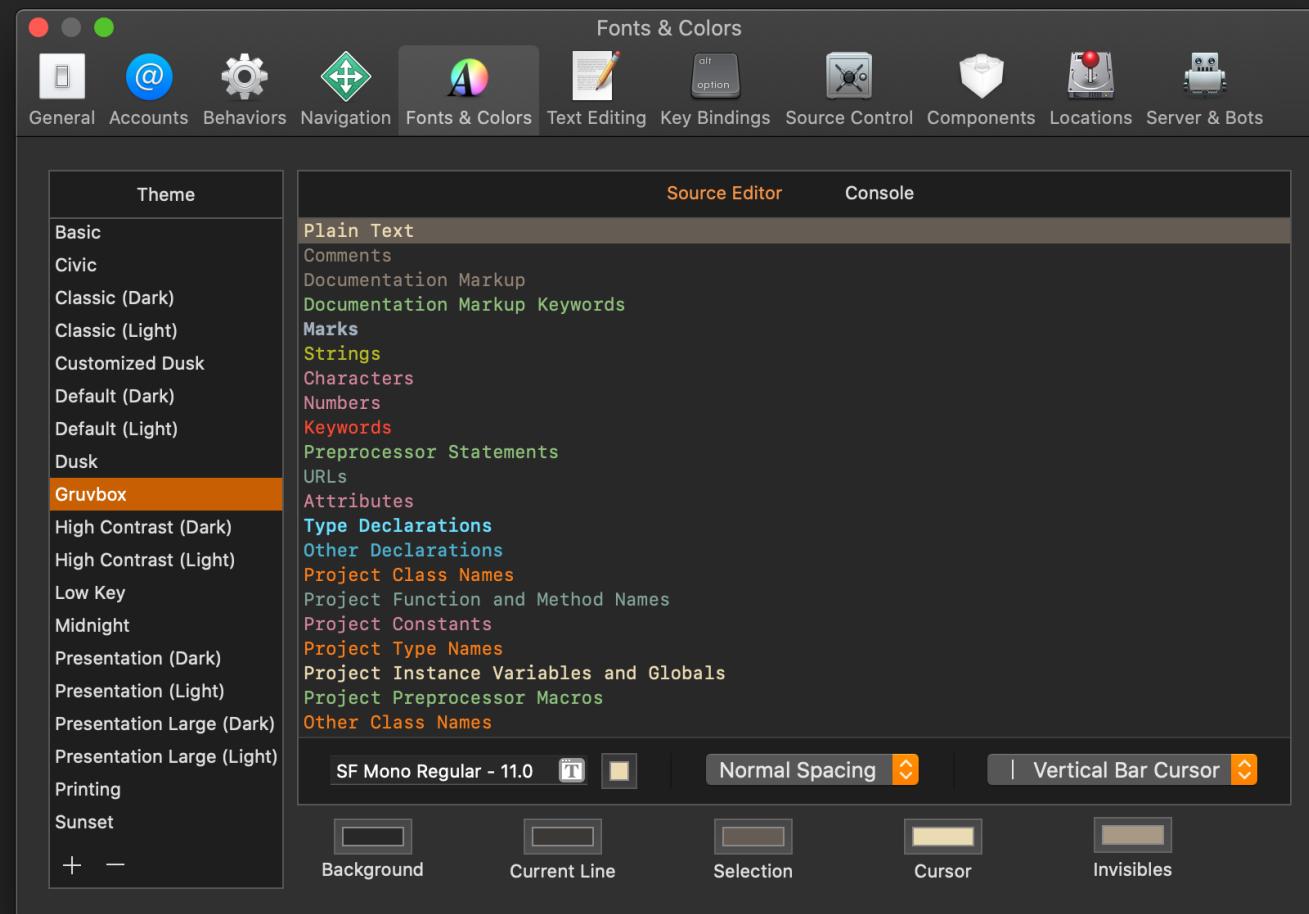
Note: In some ePUB readers you might have to
tap TWICE (2) to play the video.

Custom Code Color Theme



I created a code color theme based off of another color theme called "Gruvbox".

If you like this color theme and would like to use it in Xcode then you can [find it on my GitHub as a gist here](#).



SWIFTUI



Basic Concepts



If you are absolutely new to SwiftUI, you should definitely read through this chapter to establish some basic concepts that you can think with.



Views And Modifiers

In SwiftUI, you build a UI with **Views** and then you change those views with **Modifiers**.

View

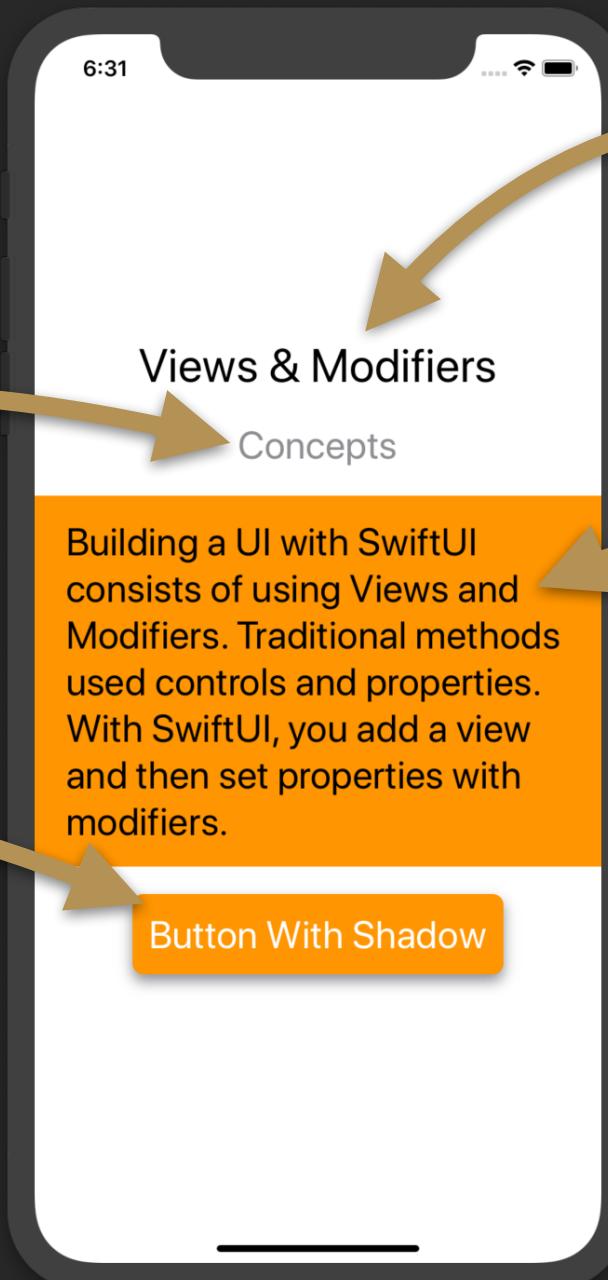
Modifiers:

- Title text size
- Gray text color

View

Modifiers:

- Title text size
- White text color
- Orange background color
- Rounded corners
- Shadow



View

Modifiers:

- Large title text size

View

Modifiers:

- Title text size
- Orange background color
- Stretched to fit device width



Containers - Vertical Layout Container

Views can be organized in containers. Some containers organize views in one direction. This is called **Stack**.

Here is an example of a **Vertical Stack** or as SwiftUI calls it, a "**VStack**".

Stacks are views too. They are views that can have modifiers applied to them.

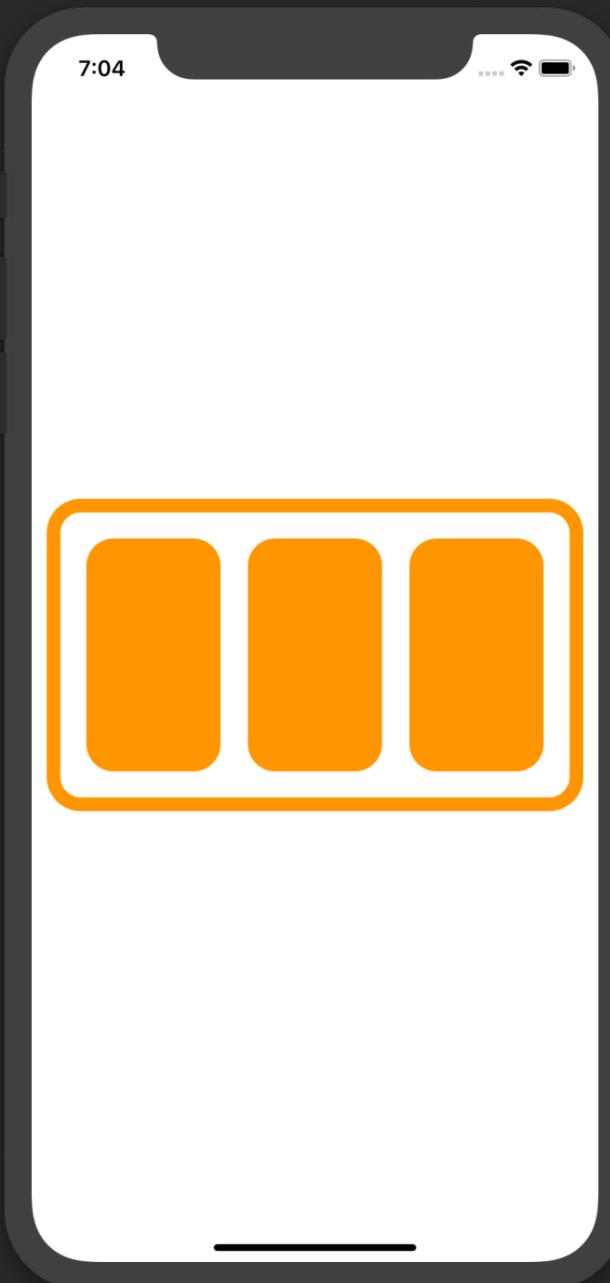




Horizontal Layout Container

There is another stack that can organize views horizontally.

SwiftUI calls this horizontal stack an **HStack**.





Depth Layout Container

Another stack view will organize your views so they are one on top of another.

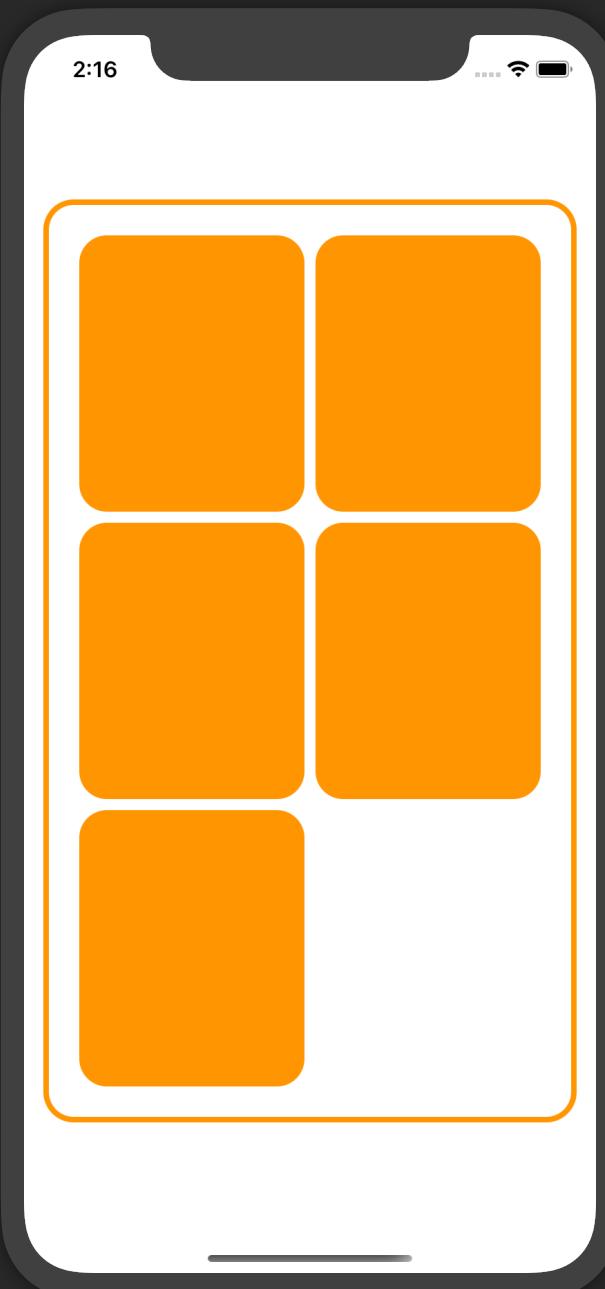
This is called the Depth Stack or **ZStack**.



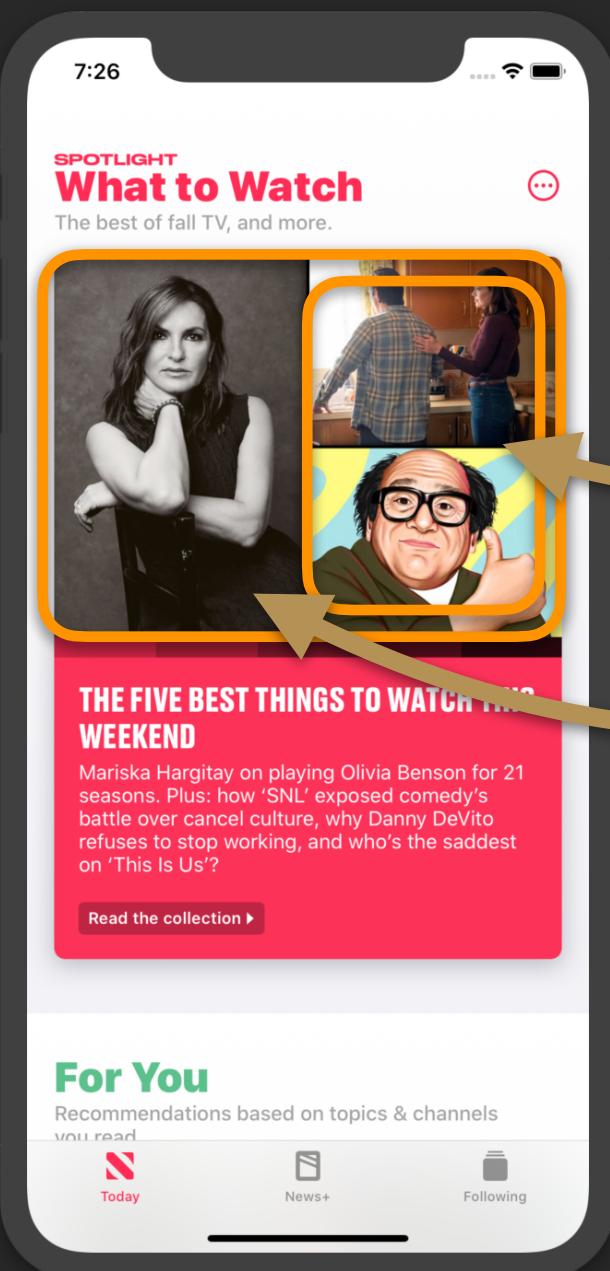


Grid Layout Container

In the second version of SwiftUI, the grid container view was introduced. There is one for horizontal and vertical layouts.



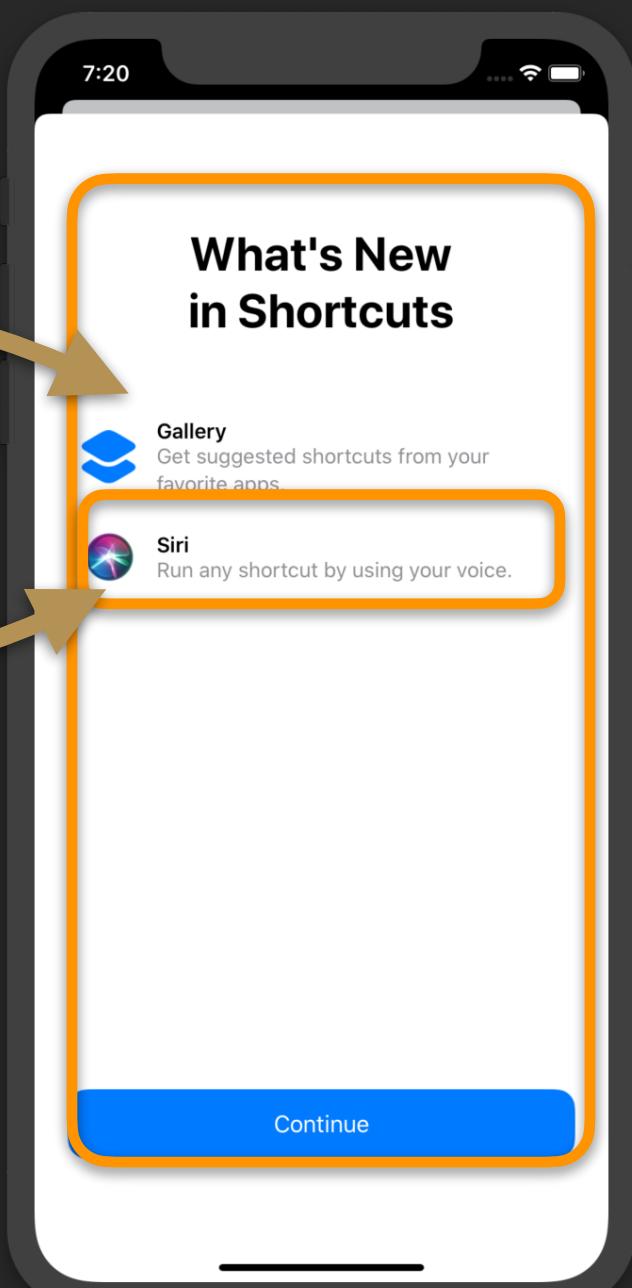
Layout Examples



Now that you know these layout stacks, you can start to guess how views like these might be arranged using SwiftUI.

VStack

HStack



**What's New
in Shortcuts**

Gallery
Get suggested shortcuts from your favorite apps.

Siri
Run any shortcut by using your voice.

Continue

In this book, you will be seeing hundreds of layout examples. Pretty soon, it will become a natural ability of yours to recognize layout.



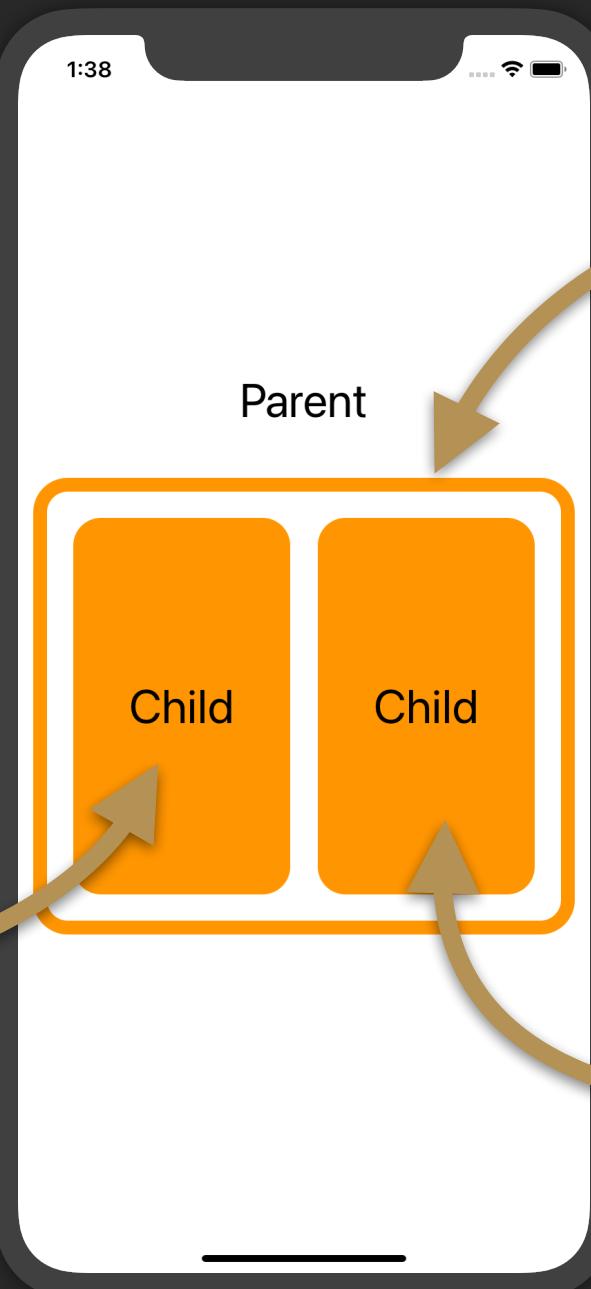
Relationships - Parent & Child

It is common in programming to express a hierarchy of objects as a parent and child relationship.

In this book, I use this concept to express relationships with SwiftUI views.

In this example, you have an HStack view that contains two child views within it. The HStack is the parent.

Child Views inside



Parent View (HStack)

It is beneficial to know that Apple refers to child views that have no children of their own as "**leaf views**".

These two child views are leaf views because they contain no other views within them.



Relationships - And Modifiers

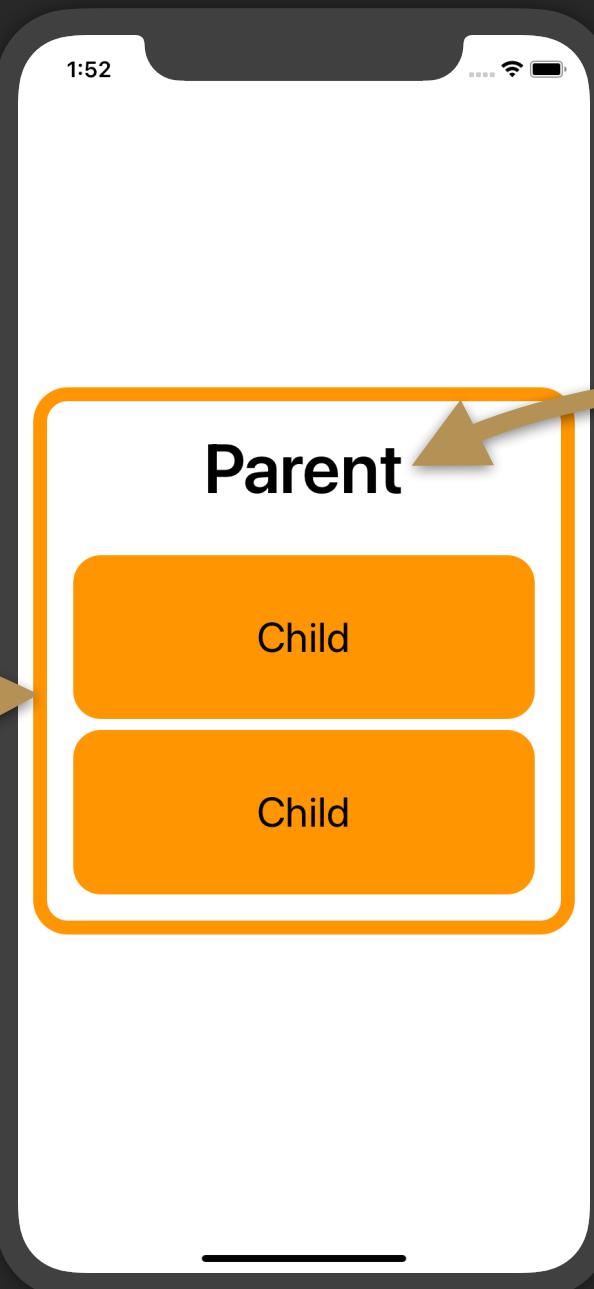
Some modifiers can be set on the parent view and it will also apply to all children in the container.
In this example, the font size is set on the parent and the child views use it.

The "Parent" text does not use the font size because it overrides it with a larger font size.

VStack View (Parent)

Modifiers applied to all child views:

- Font size of 30 points



Text View (Child)

Overriding Modifiers:

- Font size of 50 points

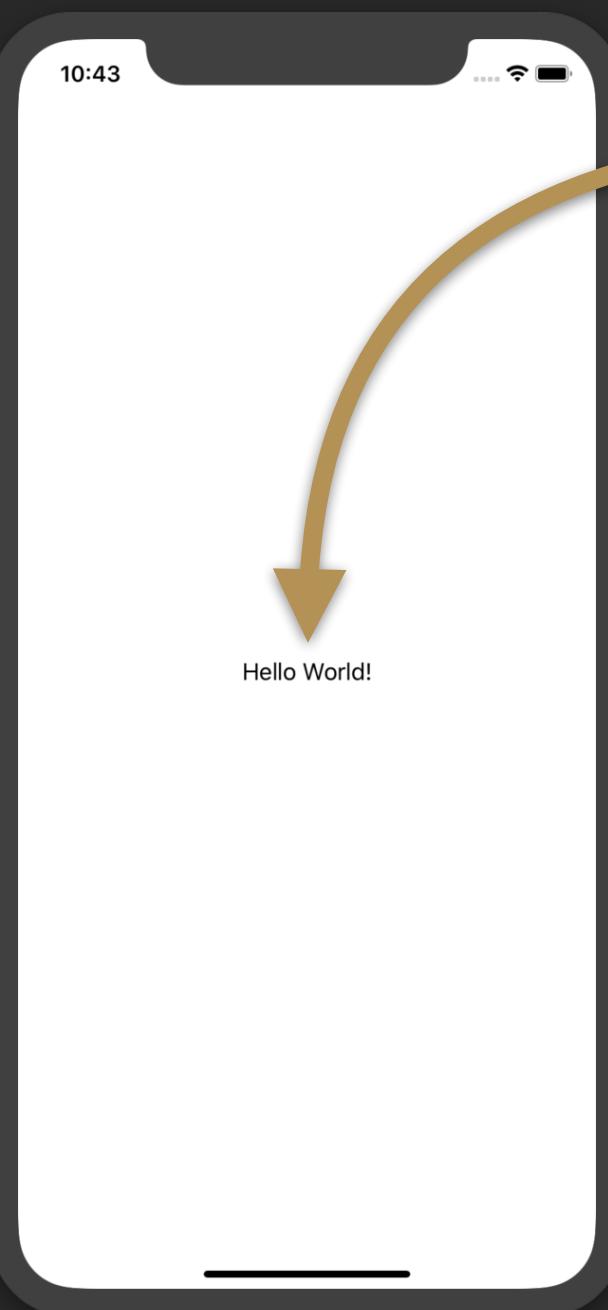
Understanding the Syntax



If you have used Swift in the past, then the SwiftUI syntax may look a little different.

It may not be readily apparent just how this code can even compile. This chapter is to help you understand how the code is able to work.

The View



```
struct BasicSyntax: View {  
    var body: some View {  
        Text("Hello World!") // Adds a text view to the screen  
    }  
}
```

Views in SwiftUI are structs that conform to the View protocol.

There is just one property to implement, the **body** property.

If “body” is a property then where is the “get” and the “return” syntax?



Property Getters

```
struct Person {
    // Computed read-only property (no setter, value is not stored)
    var personType: String {
        get {
            return "human"
        }
    }
}
```

// Change 1 – Remove the return

```
struct Person {
    var personType: String {
        get {
            "human"
        }
    }
}
```

// Change 2 – Remove the get

```
var personType: String {
    "human"
}
```

Properties can have a getter and setter. But when a property has no setter, it's called a **"read-only"** property. And when the property does not store a value, it is called a **"computed"** property. This is because the value is computed or generated every time the property is read.

In this example, `personType` is a **computed read-only** property.

You can further simplify this property in two ways:

- When the code inside the `get` is a single expression (one thing), the getter will just return it automatically. You can remove `return`.

See "*Change 1*" in the code example.

- When a property is read-only (no setter), we can remove the `get`.

Just know that these changes are **optional**.

Now when looking at this property again, you can better understand and see that it is written without the extra `get` and `return` keywords.

```
struct BasicSyntax: View {
    var body: some View {
        Text("Hello World!")
    }
}
```



SwiftUI With Property Getters



10:43



Hello World!

Since these property changes are optional, you can, for example, write the previous SwiftUI syntax with a `get` and `return` inside the `body` property. This might look more familiar to you now:

```
// SwiftUI with the get and return keywords
struct BasicSyntax: View {
    var body: some View {
        get {
            return Text("Hello World!")
        }
    }
}
```

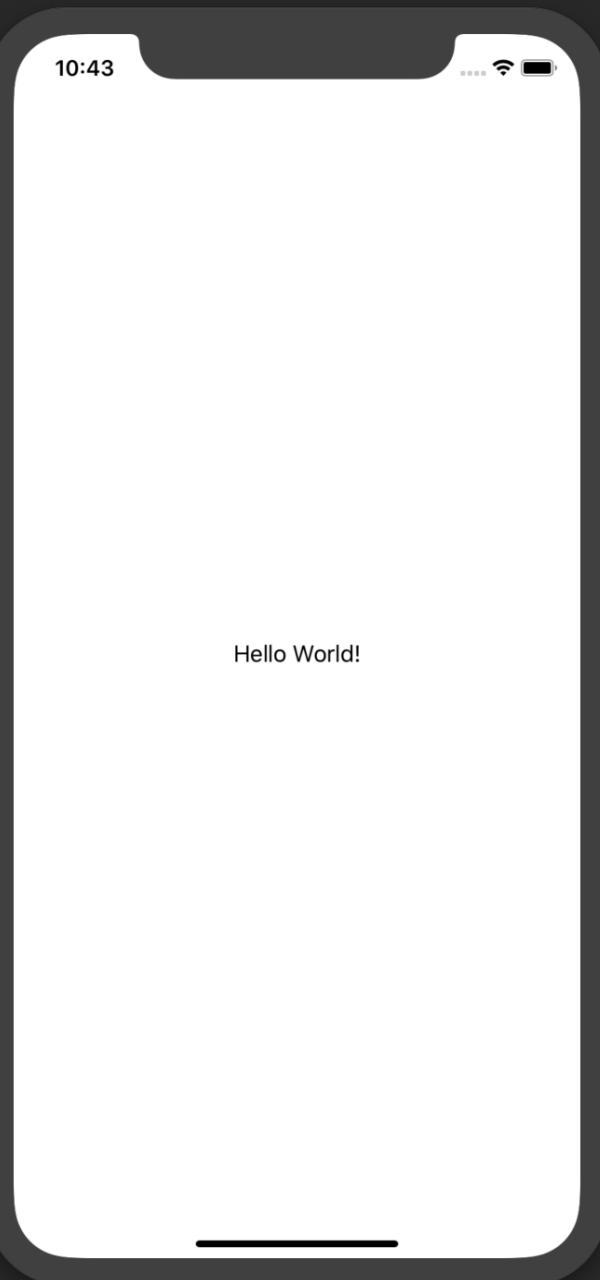


Looking at this code again, you notice the `some` keyword here. Normally, when defining a type for a property, you wouldn't see this word.

So, what does the `some` keyword do?

Opaque Types

```
struct BasicSyntax: View {  
    var body: some View {  
        Text("Hello World!")  
    }  
}
```



Opaque Types

The keyword `some` is specifying that an opaque type is being returned. In this case, the opaque type is `View`.

So why is the type called “opaque”? Well, the English definition for the word “opaque”, when referring to languages, means “hard or impossible to understand.” And this is true here because opaque types hide the value’s type information and implementation details. This will certainly make it “hard or impossible to understand” but still usable because at least you know it is a view.

When this View (`BasicSyntax`) is used by iOS to draw the screen, it doesn’t have to know that, in this example, the type `Text` is being returned. It is OK with just knowing that `some View` is being returned and can use it to draw the screen.

And so you can return anything in that `body` property as long as it conforms to the `View` protocol.

For more information on Opaque Types, I recommend referring to the Swift Programming Language documentation.



Understanding Opaque Types

Let's say you have a box of books.

You know there are books inside, but you don't know exactly what books they are. You can't see the titles of the books.

That is like an opaque type in Swift.

It hides the details of what is inside, but you know it is there, you know what kind it is and you can use it.

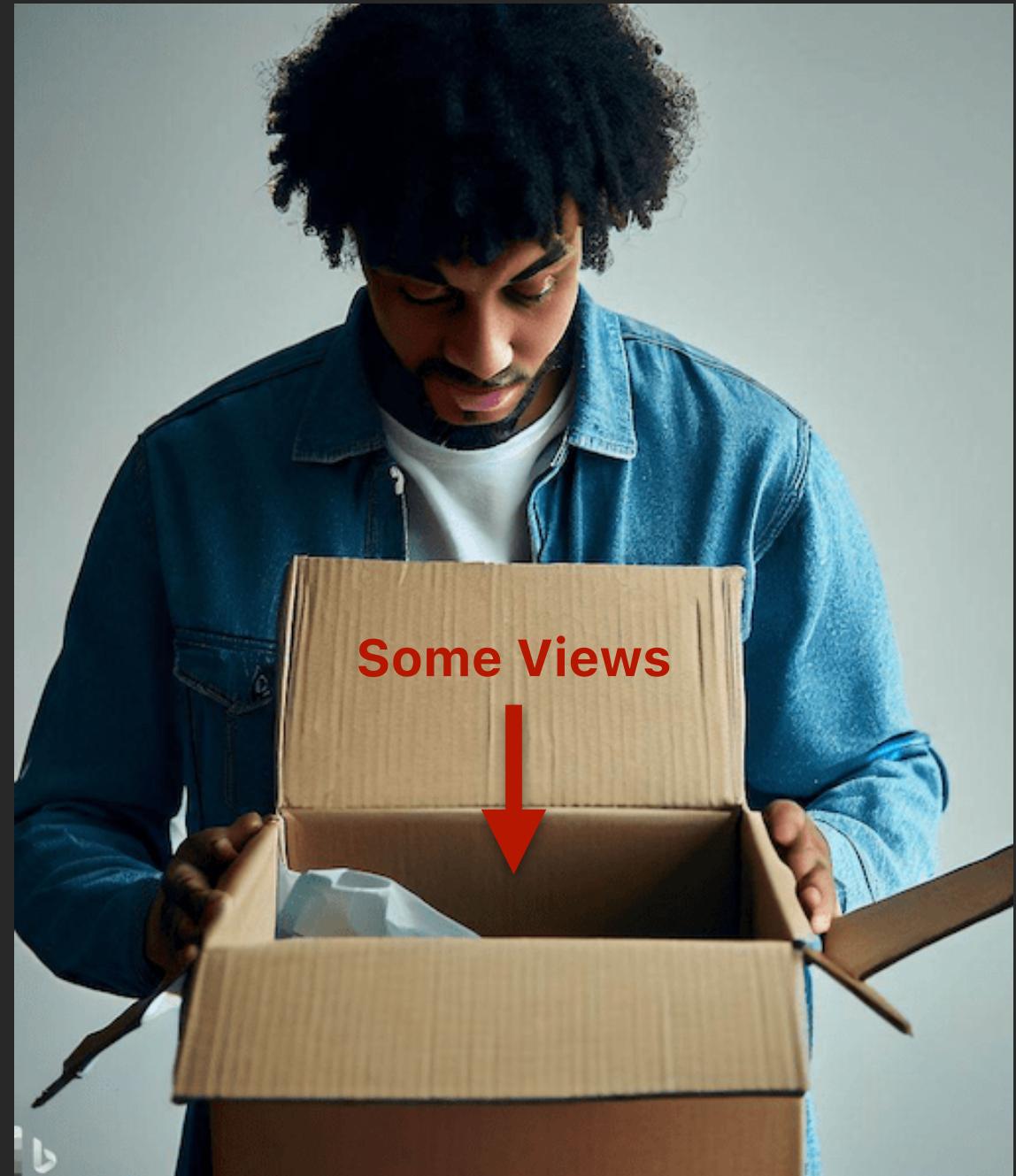
Box of Views

In this case, you have a box of different views.

They could be text, images, views for layout, etc.

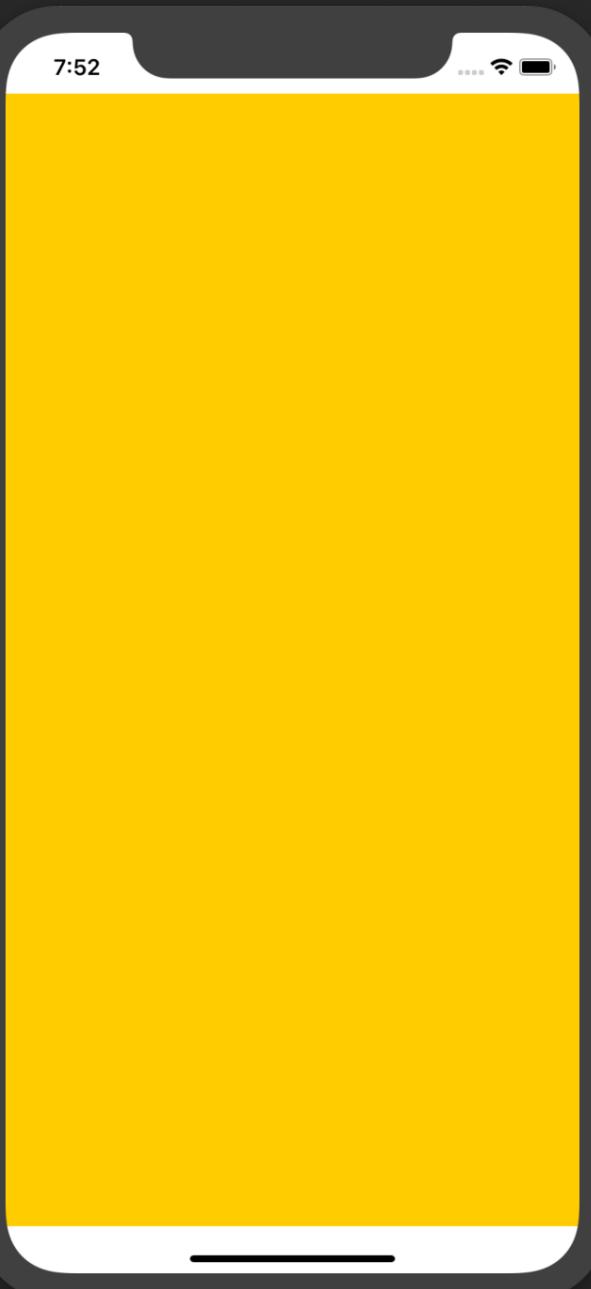
You don't care which specific type it is as long as it is a View.

There is another important thing to know about opaque types too...





Opaque Types (some Keyword)



You already know from the previous page that what is returned from the `body` property is something that conforms to the `View` protocol.

But what you also need to know is when returning an opaque type (using the `some` keyword), is that **all possible return types must all be of the same type**.

In most cases you are only returning one type. But you might have a scenario like this:

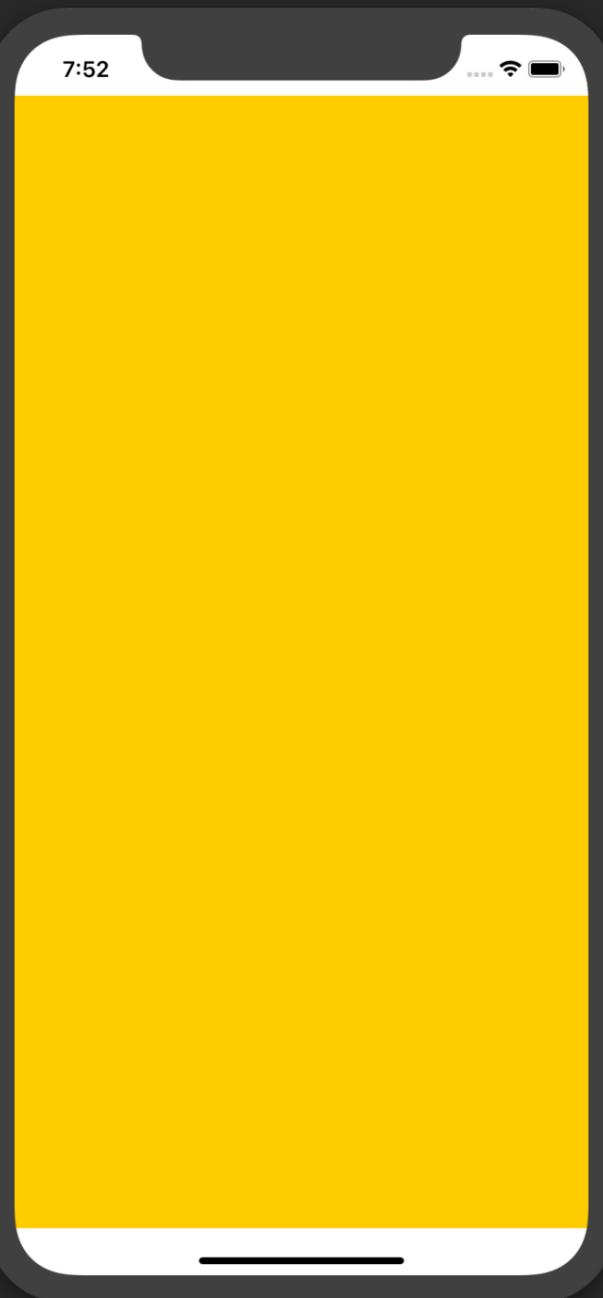
```
struct UnderstandingTheSomeKeyword: View {  
    var isYellow = true  
  
    // The keyword "some" tells us that whatever we return, it has to:  
    // 1. Conform to the View protocol  
    // 2. Has to ALWAYS be the same type of View that is returned.  
    var body: some View {  
  
        // ERROR: Function declares an opaque return type, but the return statements  
        // in its body do not have matching underlying types  
        if isYellow {  
            return Color.yellow // Color type does not match the Text type  
        }  
        return Text("No color yellow") // Text type does not match the color type  
    }  
}
```



The body property returns a `Color` and a `Text` type. This violates the `some` keyword.



Opaque Types Solution



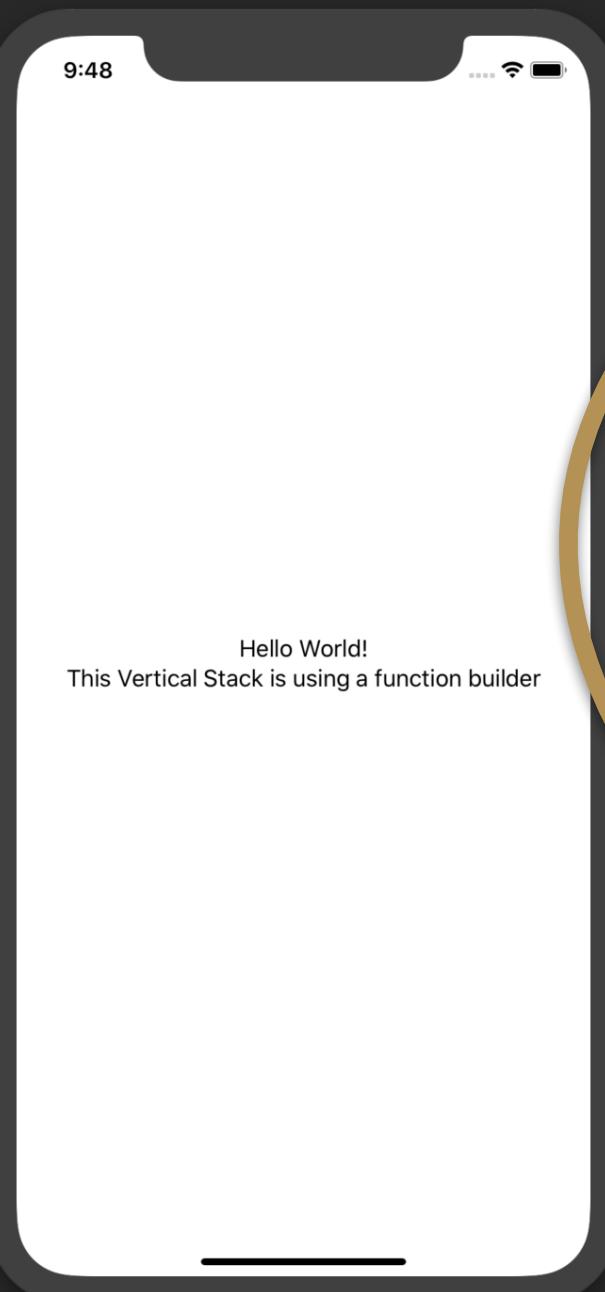
The solution would be to change the views returned so they are **all the same TYPE**. The body now returns the same type of view (Color).

```
struct UnderstandingTheSomeKeywordSolution: View {  
    var isYellow = true  
  
    // The keyword "some" tells us that whatever we return, it has to:  
    // 1. Conform to the View protocol  
    // 2. Has to ALWAYS be the same type of View that is returned.  
  
    var body: some View {  
  
        if isYellow {  
            return Color.yellow  
        }  
  
        return Color.clear  
    }  
}
```



*Now, the body property always returns a **Color** type.
This satisfies the **some** keyword.*

View Containers



```
struct Example: View {  
    var body: some View {  
        VStack {  
            Text("Hello World!")  
            Text("This Vertical Stack is using a function builder")  
        }  
    }  
}
```

So far, you have learned that `body` is a computed read-only property and can only return ONE object that is `some View`. What if you want to show multiple views though?

You learned earlier about the concept of “container” views. These are views that can contain other views. Remember, the `body` property can only return **one** view. You will get an error if you try to return more than one view in the `body` property.

In the example above, the `VStack` (Vertical Stack) is that one view being returned. And that vertical stack is a container with two more views inside of it.

The `VStack` is using a “trailing closure,” which just means that it is a code block that is passed into the initializer to be run by the `VStack`. You have probably seen this before in Swift, this is not new.

What is new in Swift is the ability to create multiple, new views within the constructor like this. Before we get into this though, let’s better understand how this constructor works.



View Container Initialization

In Swift, you usually see parentheses during initialization but **with a trailing closure, the parentheses are optional.**

You can add them and the code will still work just fine.

See “*Change 1*” in the code example.

This change may start looking more familiar to you.
Now, the question is:

How does the VStack know how to accept the multiple views like this?

This is new in Swift. To better understand this, take a look at the VStack’s initializer.

The alignment and spacing parameters are optional, that is why you don’t see them in the examples above. But notice before the content parameter there is **@ViewBuilder** syntax.

This is what allows you to declare multiple views within the content parameter’s closure.

```
struct Example: View {
    var body: some View {
        VStack {
            Text("Hello World!")
            Text("This Vertical Stack is using a function builder")
        }
    }
}
```

```
// Change 1 – Add parentheses and parameter name
struct Example: View {
    var body: some View {
        VStack(content: {
            Text("Hello World!")
            Text("This Vertical Stack is using a function builder")
        })
    }
}
```

```
// VStack initializer
init(alignment: HorizontalAlignment = .center,
     spacing: CGFloat? = nil,
     @ViewBuilder content: () -> Content)
```



@ViewBuilder Parameter Attribute

The `@ViewBuilder` parameter attribute allows Swift to build multiple child views from within a closure.

How many child views can I build within a closure?

The way this functionality is set up, you can only initialize a maximum of ten (10) views. In the example here, you will get an error because of the 11th view.

What if I need more child views?

If you need to declare more child views for your user interface, then you will have to use another view container, such as another `VStack`. (You will be seeing more options for containers in this book.)

In the second example, I use another `VStack` to contain text views 10 and 11.

```
struct ViewBuilderExample: View {  
    var body: some View {  
        VStack {  
            Text("View 1")  
            Text("View 2")  
            Text("View 3")  
            Text("View 4")  
            Text("View 5")  
            Text("View 6")  
            Text("View 7")  
            Text("View 8")  
            Text("View 9")  
            Text("View 10")  
            Text("View 11") // Will cause an error  
        }  
    }  
}  
  
struct ViewBuilderExample: View {  
    var body: some View {  
        VStack {  
            ... // Text views 1 – 5  
            Text("View 6")  
            Text("View 7")  
            Text("View 8")  
            Text("View 9")  
            VStack { // The VStack is now the 10th view  
                Text("View 10")  
                Text("View 11")  
            }  
        }  
    }  
}
```

Only 10 views allowed.

My Template

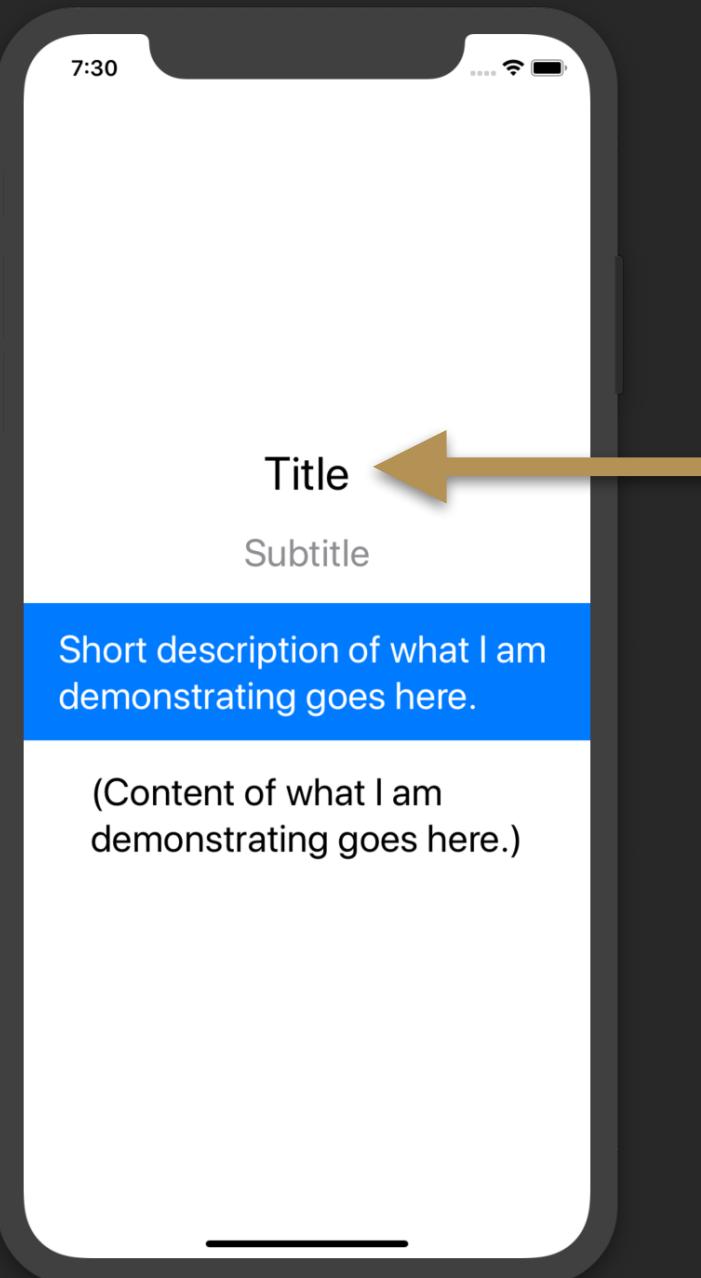


If you are completely new to SwiftUI you may wonder what a lot of this code means right at the beginning of the book. I have "templates" that contains a title, subtitle and a short description on most SwiftUI screens.

I will take you through step-by-step on how I build this template that I use throughout the book. I will describe each one only briefly because each modifier I apply to the views here are described in more detail throughout the book within their own sections.



My Basic Template



Here is my basic template I use throughout the book to explain views and modifiers.

In the next pages I'm going to explain how this is built in SwiftUI. I want to make sure you understand these parts because you will see them everywhere in this book.

I want to remove any confusion right at the beginning so it doesn't get in your way to learning the topics in the book.

Let's start with the title.



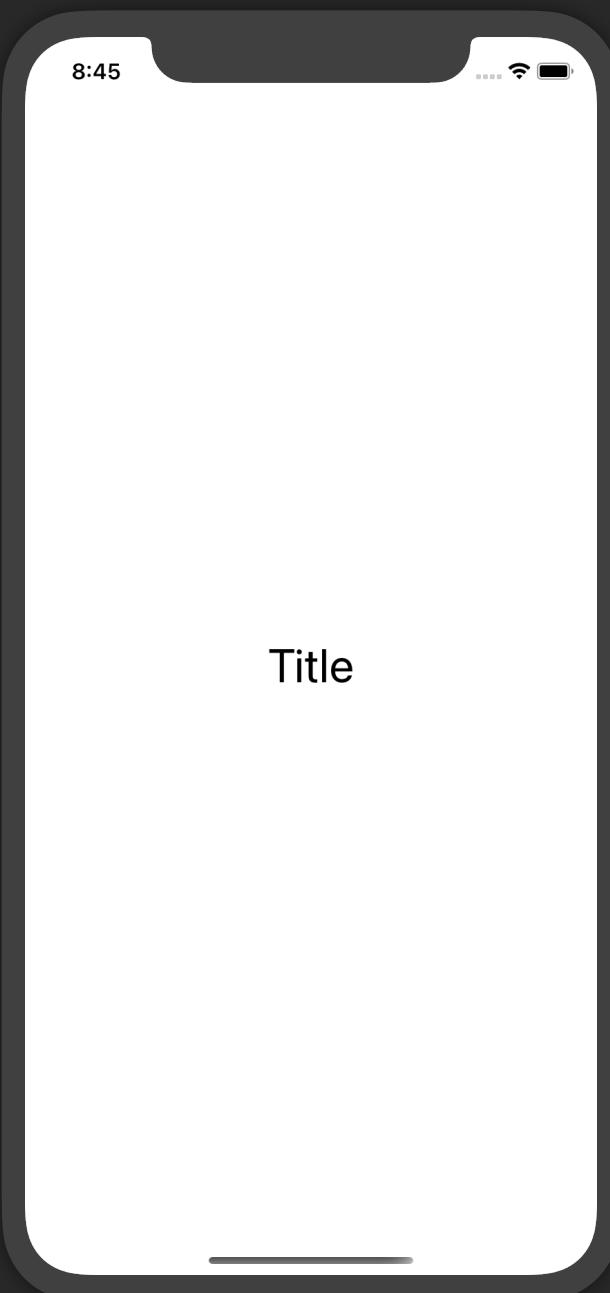
Starting with the Title

```
struct Title: View {  
    var body: some View {  
        Text("Title") // Create text on the screen  
        .font(.largeTitle) // Use a font modifier to make text larger  
    }  
}
```



Here, you have a **Text** view. You want to make it larger so you use the **font** modifier so you can set the size to a SwiftUI preset size called **largeTitle** (this is the largest preset size).

There are more ways you can change the size of text that are covered in this book in the **Control Views** chapter, in the section called **Text**.





Add a VStack

```
struct AddVStack: View {  
    var body: some View {  
        // Only one view can be returned from the body property.  
        // Add 20 points between views within this container.  
        VStack(spacing: 20) { // VStack is a container view that can hold up to 10 views  
            Text("Title")  
                .font(.largeTitle)  
        }  
    }  
}
```

Title

VStack

The `body` property can **only return one view**. You will get an error if you have two views.

So, you need to use a container view that will contain multiple views. The vertical stack (`VStack`) is the perfect choice here.

Now you can add up to 9 more views to the `VStack`.

Spacing

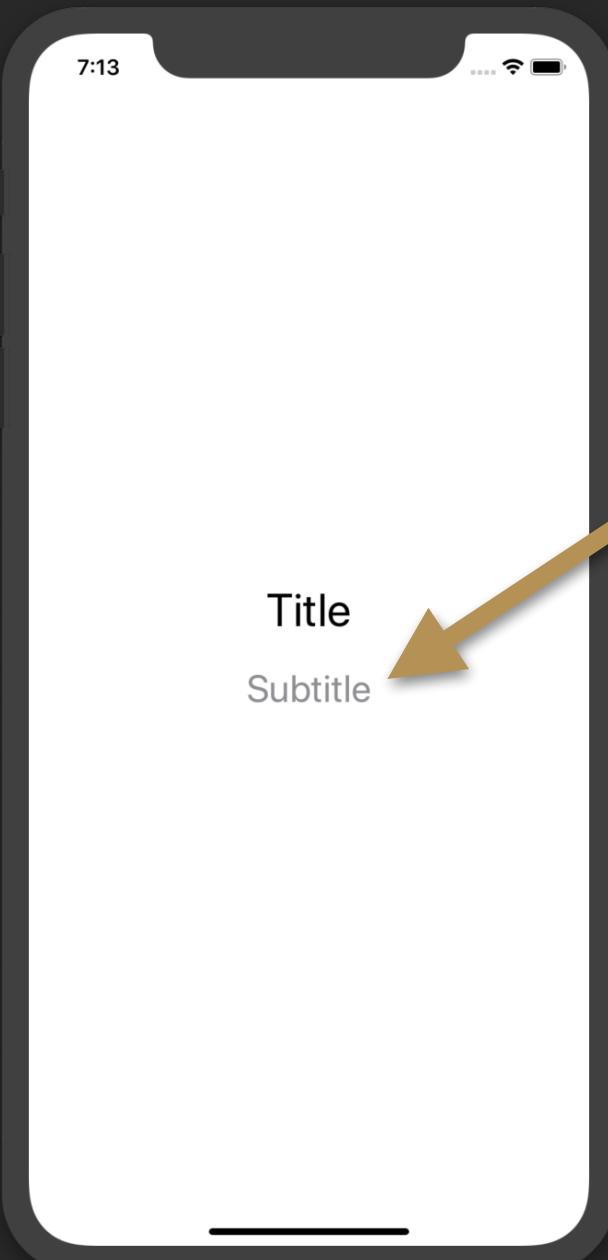
The `VStack` has an optional parameter you can use in its initializer to specify how many points of spacing you want in between views. (**Note:** `spacing` does not add spacing to the top or bottom of the `VStack`.)

Now, let's add the subtitle text.



Adding the Subtitle

```
struct Subtitle: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Title")  
                .font(.largeTitle)  
  
            Text("Subtitle")  
                .font(.title) // Set to be the second largest font.  
                .foregroundColor(Color.gray) // Change text color to gray.  
        }  
    }  
}
```



Subtitle

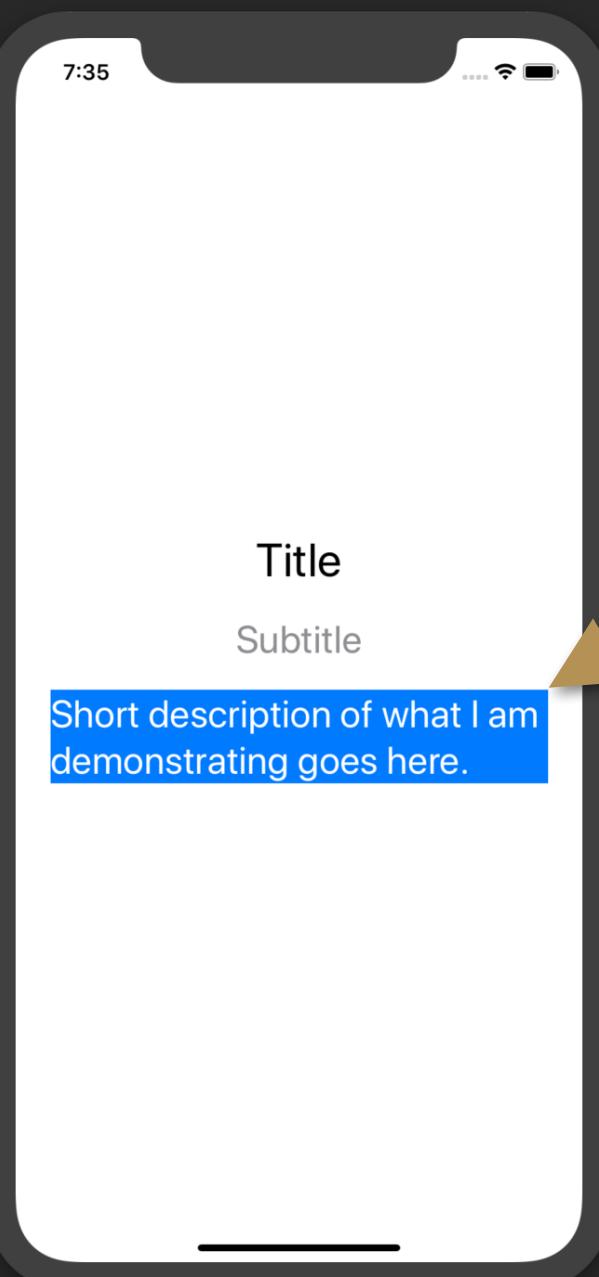
The subtitle is another text view. This time, you set the size to be the second largest preset size with the **title** parameter.

Finally, you modify the view to change the text color to gray. (**Note:** instead of using **Color.gray** you can also use just **.gray**.)

Now, let's add the description text.



Add the Description with a Background Color



```
struct Description1: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Title")  
                .font(.largeTitle)  
  
            Text("Subtitle")  
                .font(.title)  
                .foregroundColor(.gray)  
  
            Text("Short description of what I am demonstrating goes here.")  
                .font(.title)  
                .foregroundColor(Color.white)  
                .background(Color.blue) // Add the color blue behind the text.  
        }  
    }  
}
```

With the description text view, you are now familiar with the `font` and `foregroundColor` modifiers. But now you want to add a color behind the text. So you use the `background` modifier to set a color.

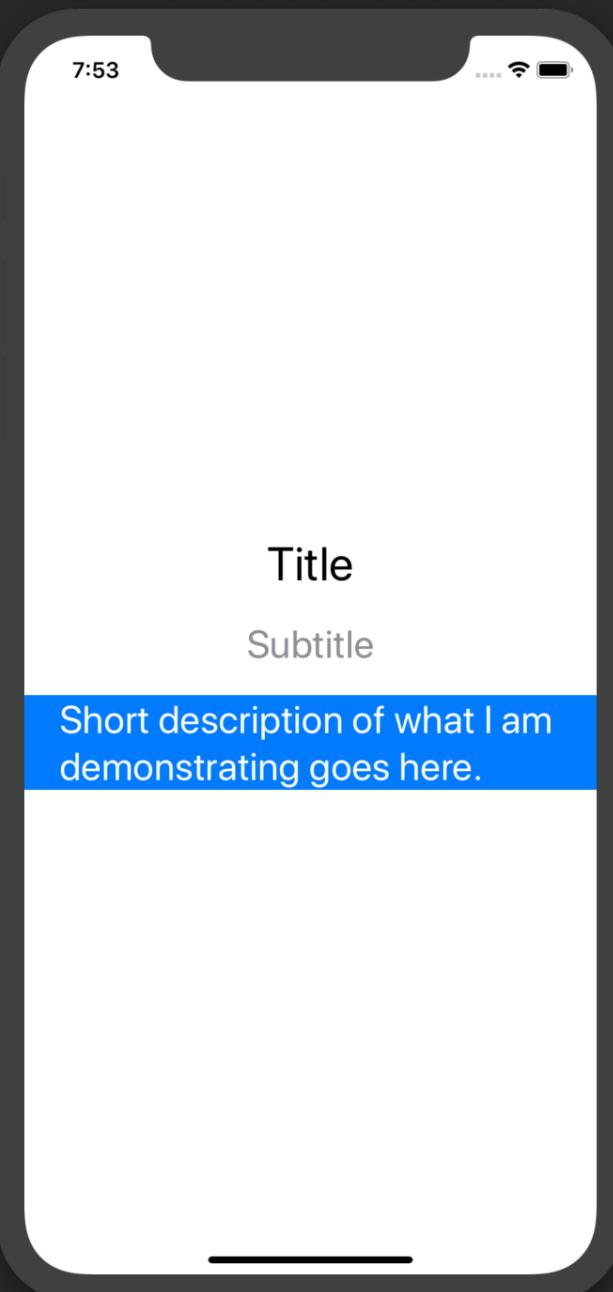
The important thing to notice here is it is not a `backgroundColor` modifier. That does not exist. It is a `background` modifier because it adds a layer **behind** the view.

`Color.blue` is actually a view. So the `background` modifier is adding a blue view on a layer behind the text.

We want this view to extend to the edges of the screen. So let's add that next.



Adding a Frame Modifier



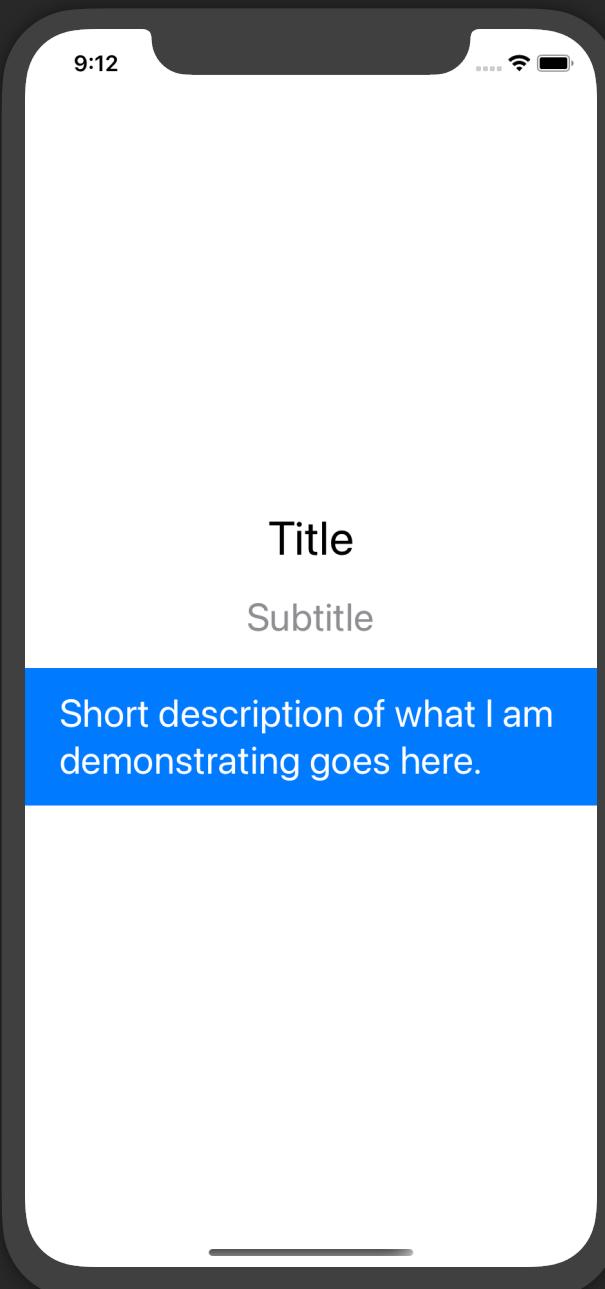
```
struct Description2: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Title")  
                .font(.largeTitle)  
  
            Text("Subtitle")  
                .font(.title)  
                .foregroundColor(.gray)  
  
            Text("Short description of what I am demonstrating goes here.")  
                .frame(maxWidth: .infinity) // Extend until you can't go anymore.  
                .font(.title)  
                .foregroundColor(Color.white)  
                .background(Color.blue)  
        }  
    }  
}
```

To extend the text to the edges of the device, we use the `frame` modifier. You don't need to set a fixed value. Instead, you can just modify the text view and say its frame's maximum width can extend to `infinity` until it hits its parent's frame and then will stop. Its parent's frame is the `VStack`.

This is looking good. It would look better though if there was more space around the text that pushed out the blue background.



Add Padding Around the Text View



```
struct Description3: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Title")  
                .font(.largeTitle)  
  
            Text("Subtitle")  
                .font(.title)  
                .foregroundColor(.gray)  
  
            Text("Short description of what I am demonstrating goes here.")  
                .frame(maxWidth: .infinity)  
                .font(.title)  
                .foregroundColor(Color.white)  
                .padding() // Add space all around the text  
                .background(Color.blue)  
        }  
    }  
}
```

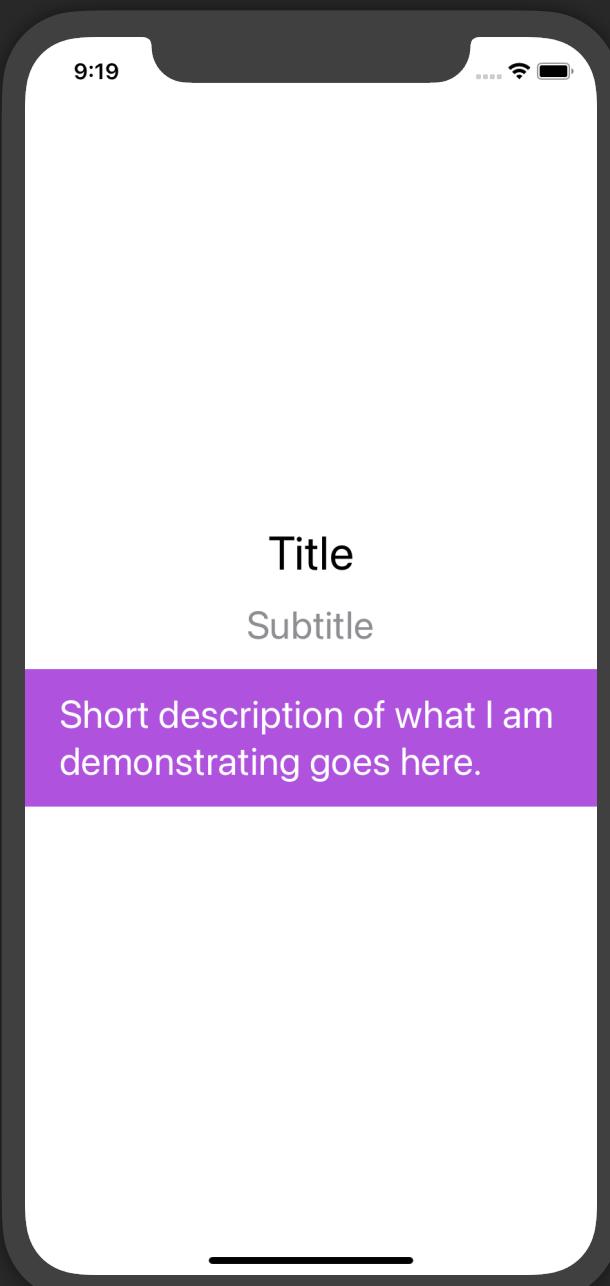
Padding

Use the `padding` modifier to add space around a view. Remember, the order of modifiers matter. You can add the `padding` modifier anywhere as long as it is BEFORE the `background` modifier. If it was after the `background`, it would add space around the blue background. We want the space between the text and the background.



Version 2 of the Template

```
struct Version2: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Title",  
                      subtitle: "Subtitle",  
                      desc: "Short description of what I am demonstrating goes here.",  
                      back: .purple, textColor: .white)  
        }  
        .font(.title)  
    }  
}
```



Version 2

When I updated the book with SwiftUI 2, I wanted a more efficient way of adding a title, subtitle and description.

So I made my own view, called `HeaderView`, where I can pass in the information and it will format it.

As you can see, this saves repetitive code and space.

If you're interested in how this is done, look in the Xcode project that comes with the paid book bundle for the file "**HeaderView.swift**".

SwiftUI Basics



Now that you understand this basic template I use for demonstrating topics, I will start using it. Be sure to read what is on each screenshot (or find the text in the code to read).



Refactoring



```
struct Refactoring: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Refactoring")  
                .font(.largeTitle)  
            Text("Reusing Modifiers")  
                .foregroundColor(.gray)  
            Text("You can put common modifiers on the parent views to be applied to all the  
child views.")  
                .frame(maxWidth: .infinity)  
                .foregroundColor(Color.white)  
                .padding()  
                .background(Color.blue)  
        }  
        .font(.title) // This font style will be applied to ALL text views inside the VStack.  
    }  
}
```

Overrides `.font(.title)`

Here, the `title` font is being applied to all three text views inside of the parent view (`VStack`).

Why isn't the title text view affected?

Because the title text view sets the font again, it overrides the `title` size with the `largeTitle` size.



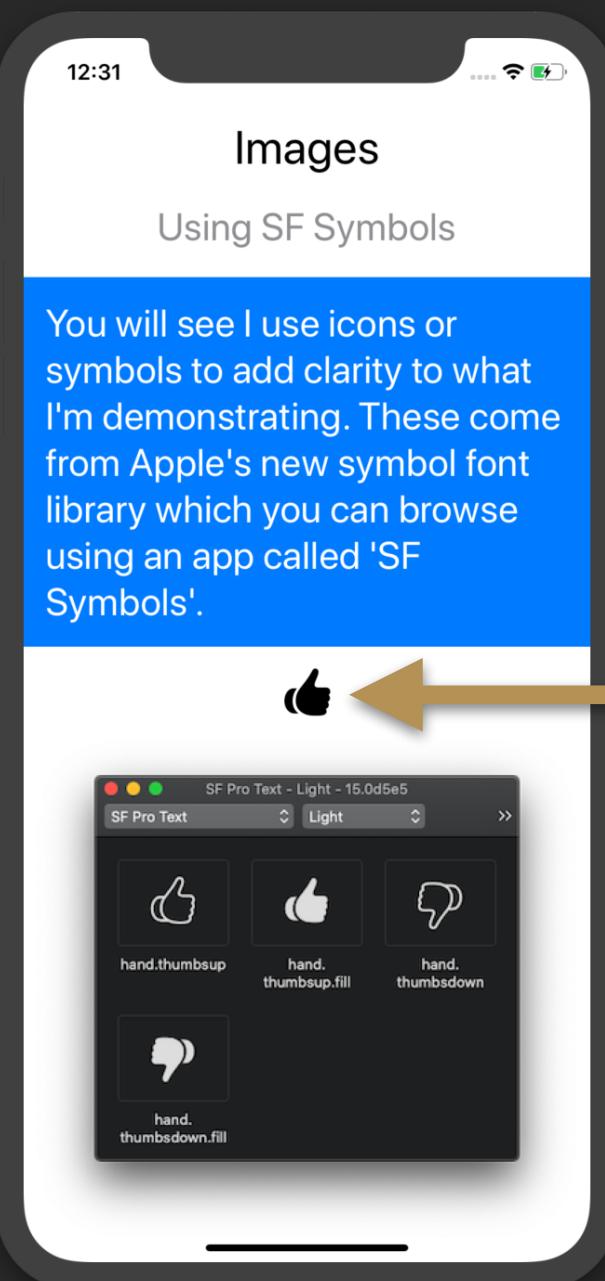
Short Introduction to Symbols

```
struct SymbolsIntro: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Images")
                .font(.largeTitle)
            Text("Using SF Symbols")
                .foregroundColor(.gray)

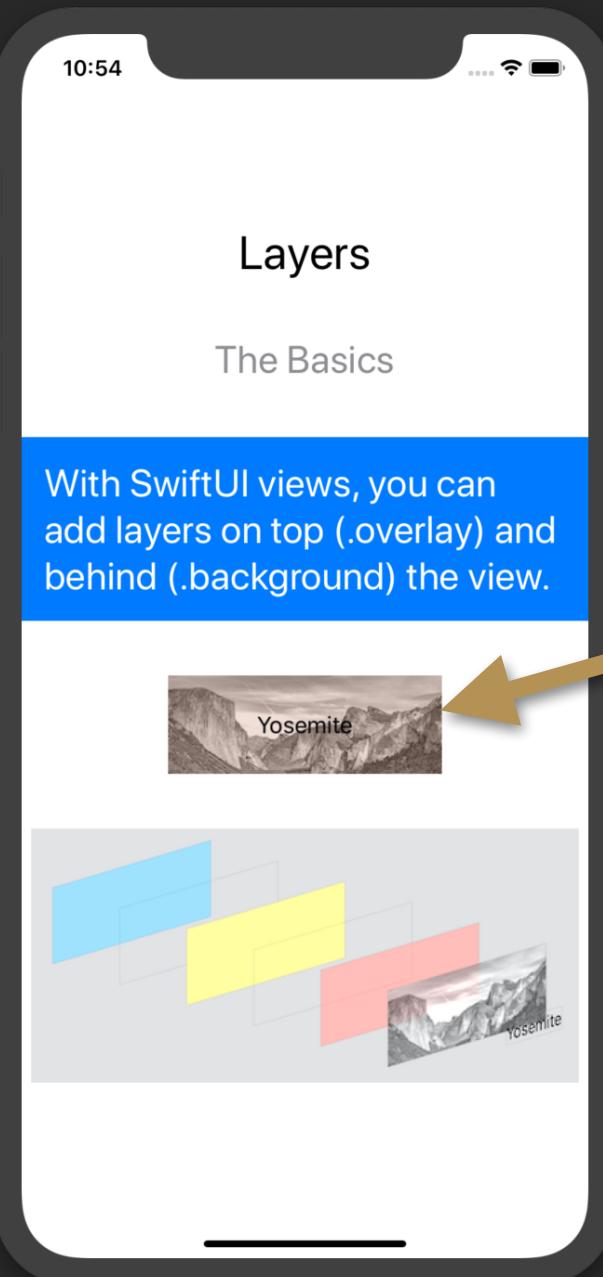
            Text("You will see I use icons or symbols to add clarity to what I'm demonstrating. These come from Apple's new symbol font library which you can browse using an app called 'SF Symbols'.")
                .frame(maxWidth: .infinity)
                .padding()
                .background(Color.blue)
                .foregroundColor(Color.white)

            // Use "systemName" when you want to use "SF Symbols"
            Image(systemName: "hand.thumbsup.fill")
                .font(.largeTitle) // Make the symbol larger
        }
        .font(.title)
        .ignoresSafeArea(edges: .bottom) // Ignore the bottom screen border
    }
}
```

Even though an **Image** view is used to initialize a symbol, you use the **font** modifier to change its size. These symbols actually come from fonts. So use font modifiers to change them. There is a whole section that covers this. Go [here to download](#) and install the SF Symbols app.



Layers



```
VStack(spacing: 40) {  
    Text("Layers")  
        .font(.largeTitle)  
  
    Text("The Basics")  
        .foregroundColor(.gray)  
  
    Text("With SwiftUI views, you can add layers on top (.overlay) and behind (.background) the view.")  
        .frame(maxWidth: .infinity)  
        .padding()  
        .background(Color.blue)  
        .foregroundColor(Color.white)  
  
    Image("yosemite") // Show an image from Assets.xcassets  
        .opacity(0.7) // Make image only 70% solid  
        .background(Color.red.opacity(0.3)) // Layer behind image  
        .background(Color.yellow.opacity(0.3)) // Layer behind red  
        .background(Color.blue.opacity(0.3)) // Layer behind yellow  
        .overlay(Text("Yosemite")) // Layer on top of image  
}  
    .font(.title)
```

I use layers (`background` and `overlay`) early in the book so I want to make sure you understand this concept.

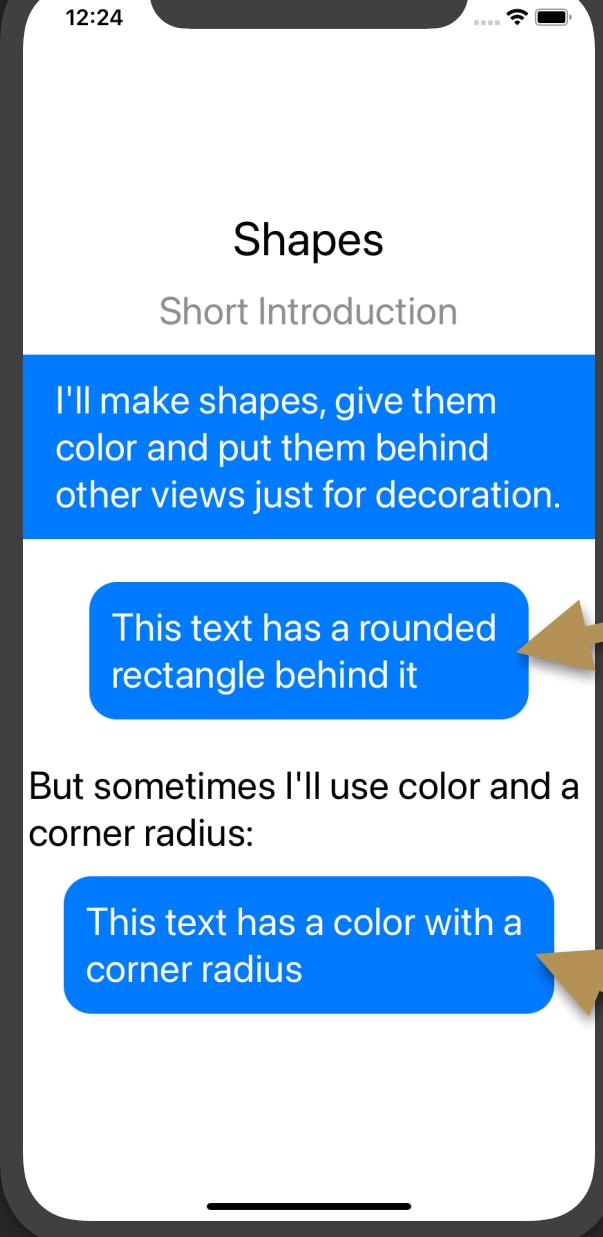
Both of these modifiers are explained in detail in their own sections.



Short Introduction to Shapes

```
 VStack(spacing: 20) {  
     Text("Shapes")  
         .font(.largeTitle)  
     Text("Short Introduction")  
         .foregroundColor(.gray)  
     Text("I'll make shapes, give them color and put them behind other views just for decoration.")  
         .frame(maxWidth: .infinity)  
         .padding()  
         .background(Color.blue)  
         .foregroundColor(Color.white)  
  
     Text("This text has a rounded rectangle behind it")  
         .foregroundColor(Color.white)  
         .padding()  
         .background(  
             RoundedRectangle(cornerRadius: 20) // Create the shape  
                 .foregroundColor(Color.blue)) // Make shape blue  
         .padding()  
  
     Text("But sometimes I'll use color and a corner radius:")  
  
     Text("This text has a color with a corner radius")  
 }
```

RoundedRectangle is a common shape.



Layout Behavior



In SwiftUI, you may wonder why some views layout differently than others. You can observe two behaviors when it comes to the size and layout of views:

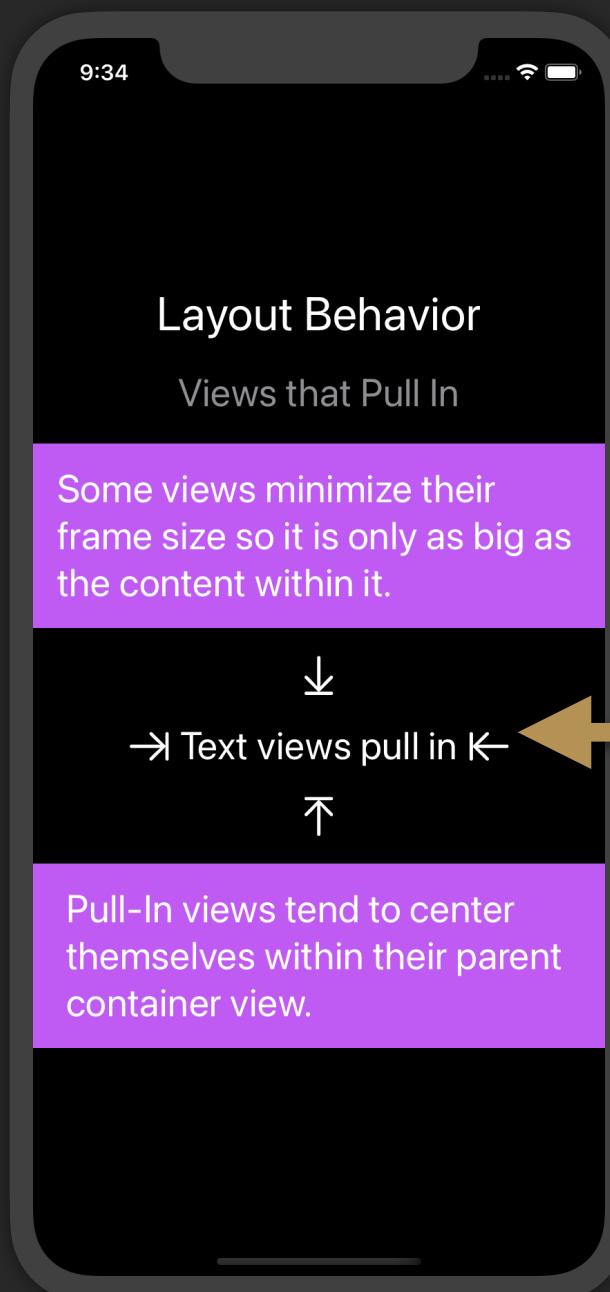
1. Some views pull in to be as small as possible to fit their content. (I will refer to these as “pull-in” views.)
2. Some views push out to fill all available space. (I will refer to these as “push-out” views.)

Knowing these two behaviors can help you predict what will happen when using the different views.



Some Views Pull In

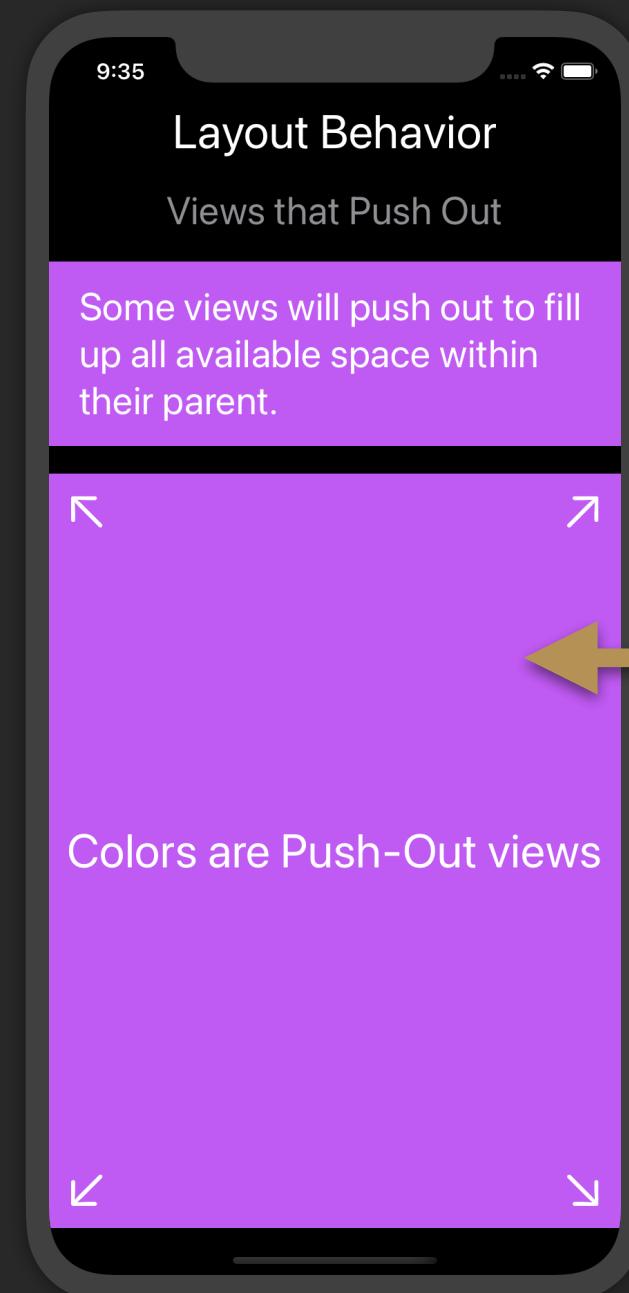
```
struct ViewSizes_Pull_In: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Layout Behavior").font(.largeTitle)  
            Text("Views that Pull In").foregroundColor(.gray)  
            Text("Some views minimize their frame size so it is only as big as the  
                content within it.")  
            .frame(maxWidth: .infinity)  
            .padding()  
            .background(Color.purple)  
            .foregroundColor(Color.white)  
  
            Image(systemName: "arrow.down.to.line.alt")  
  
            HStack { // Order views horizontally  
                Image(systemName: "arrow.right.to.line.alt")  
                Text("Text views pull in")  
                Image(systemName: "arrow.left.to.line.alt")  
            }  
  
            Image(systemName: "arrow.up.to.line.alt")  
  
            Text("Pull-In views tend to center themselves within their parent container  
                view.")  
            .frame(maxWidth: .infinity)  
            .padding()  
            .background(Color.purple)  
            .foregroundColor(Color.white)  
            }.font(.title)  
    }  
}
```





Some Views Push Out

```
struct ViewSizes_Push_Out: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Layout Behavior")  
            Text("Views that Push Out")  
                .font(.title).foregroundColor(.gray)  
            Text("Some views will push out to fill up all available space within their parent.")  
                .frame(maxWidth: .infinity).padding().font(.title)  
                .background(Color.purple)  
        }  
        .font(.largeTitle) // Make all text and symbols larger  
    }  
}
```



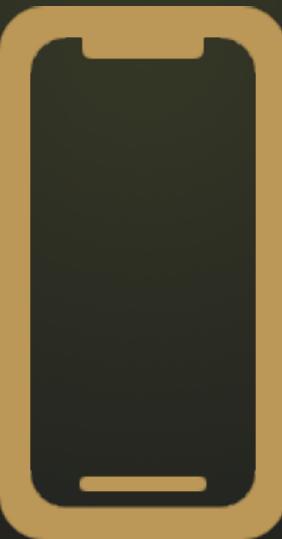
Color.purple

// Add 5 layers on top of the color view
.overlay(
 Image(systemName: "arrow.up.left")
 .padding() // Add spacing around the symbol
 , alignment: .topLeading) // Align within the layer
.overlay(
 Image(systemName: "arrow.up.right")
 .padding(), alignment: .topTrailing)
.overlay(
 Image(systemName: "arrow.down.left")
 .padding(), alignment: .bottomLeading)
.overlay(
 Image(systemName: "arrow.down.right")
 .padding(), alignment: .bottomTrailing)
.overlay(Text("Colors are Push-Out views"))
}.font(.largeTitle) // Make all text and symbols larger

Colors are push-out views.

For the most part, I will be telling you if a view is a push-out view or a pull-in view at the beginning of the sections.

SEE YOUR WORK



Preview Options



As you practice these examples, you might want to see your SwiftUI working on different devices in different modes, including light or dark mode or with different accessibility settings.

You can do all of this **without** even having to launch the Simulator.

When using SwiftUI, you get a preview canvas that will show you how your views will render.

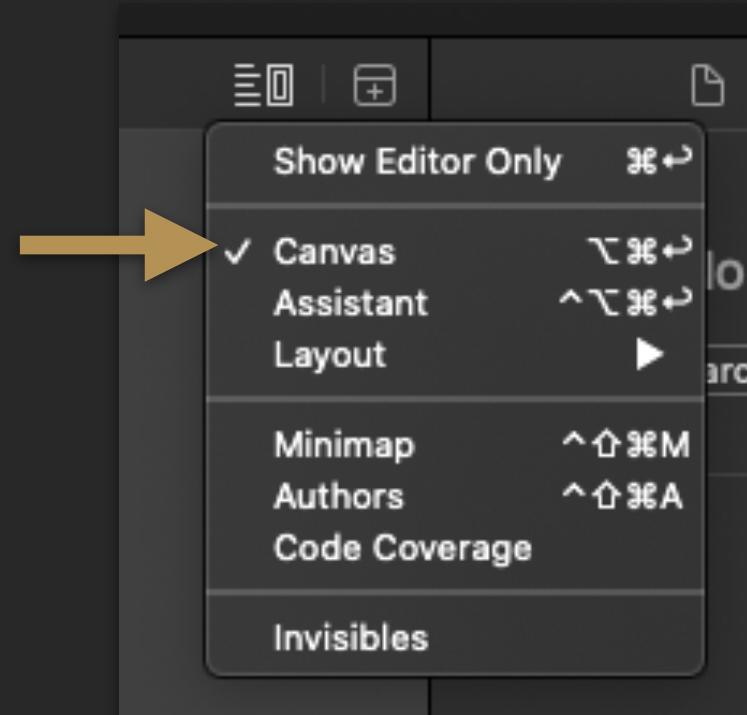
The Canvas - What is it?

Code

Preview
or
Canvas
or
Preview Canvas

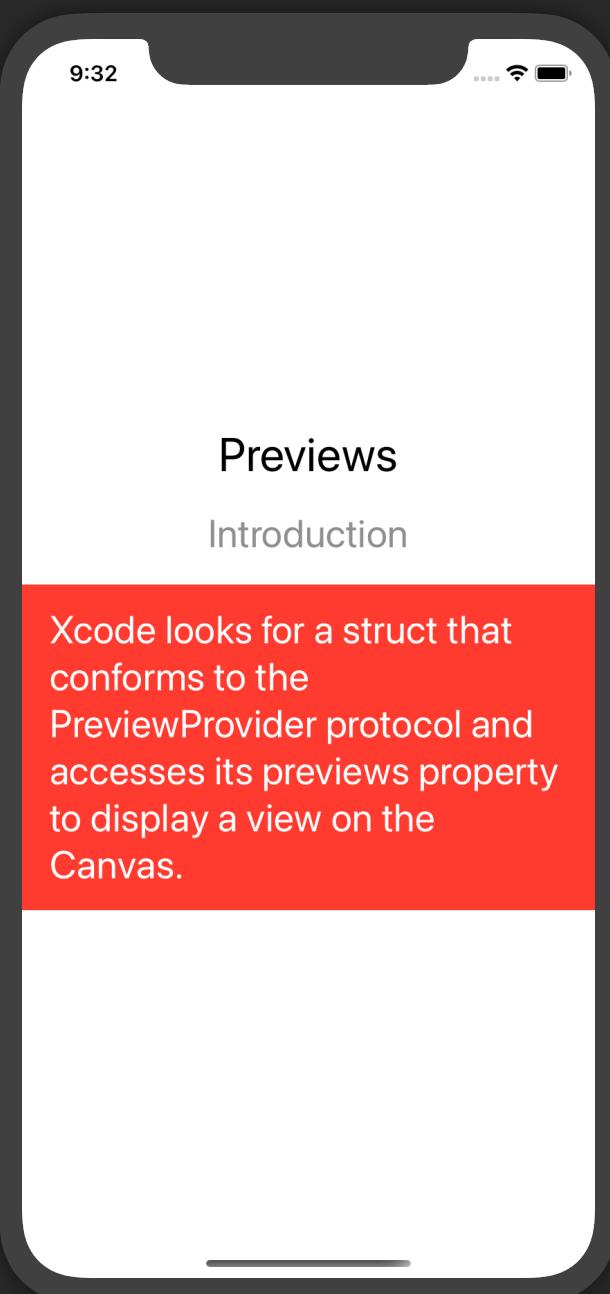
The canvas is the area next to the code that shows you a preview of how your SwiftUI will look. You might also hear this called the “**Preview**” or “**Preview Canvas**”.

If you do not see this pane, click on the Editor Options button on the top right of your code window and click Canvas:



Introduction

```
struct Previews_Intro: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Previews")  
                .font(.largeTitle)  
  
            Text("Introduction")  
                .foregroundColor(.gray)  
  
            Text("Xcode looks for a struct that conforms to the PreviewProvider  
protocol and accesses its previews property to display a view on the Canvas.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.red)  
                .foregroundColor(.white)  
            }.font(.title)  
    }  
}  
  
// Xcode looks for PreviewProvider struct  
struct Previews_Intro_Previews: PreviewProvider {  
    // It will access this property to get a view to show in the Canvas (if the  
Canvas is shown)  
    static var previews: some View {  
        // Instantiate and return your view inside this property to see a preview  
        of it  
        Previews_Intro()  
    }  
}
```



Dark Mode

```
struct Preview_DarkMode: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Previews").font(.largeTitle)  
            Text("Dark Mode").foregroundColor(.gray)  
            Text("By default, your preview will show in light mode. To see it in  
dark mode, you can use the environment modifier.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.red)  
                .foregroundColor(.white)  
        }.font(.title)  
    }  
}
```

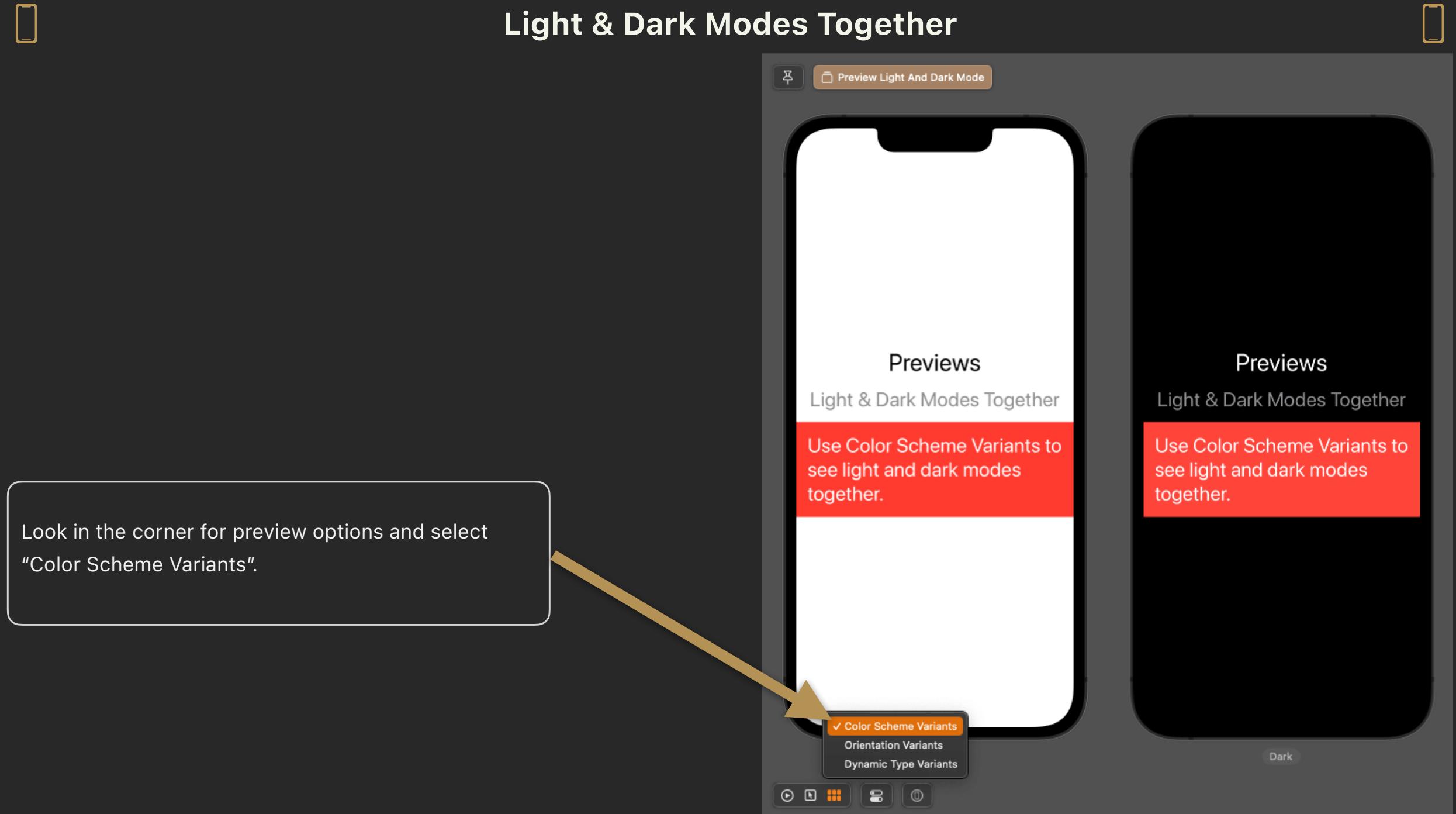
```
struct Preview_DarkMode_Previews: PreviewProvider {  
    static var previews: some View {  
        Preview_DarkMode()  
            .preferredColorScheme(.dark)  
    }  
}
```

Previews

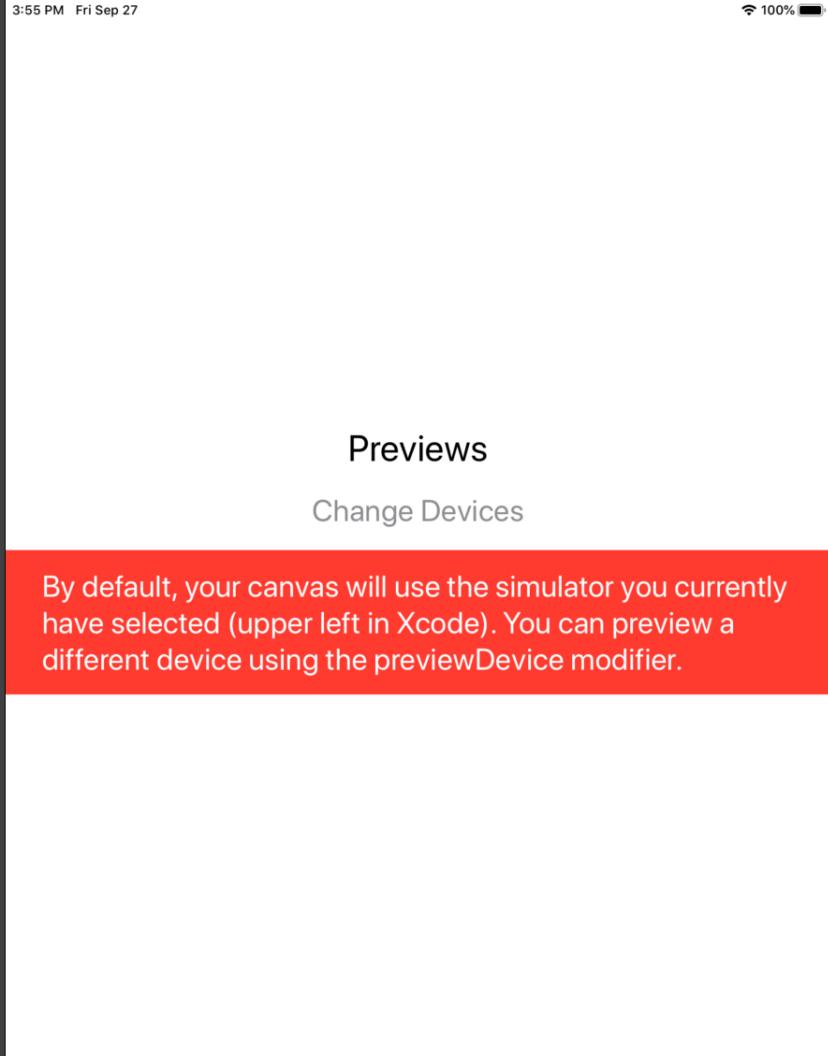
Dark Mode

By default, your preview will show in light mode. To see it in dark mode, you can use the environment modifier.

Light & Dark Modes Together



Changing Devices



```
struct Previews_Devices: View {
    var body: some View {
        VStack(spacing: 20) {
            Text("Previews").font(.largeTitle)
            Text("Change Devices").foregroundColor(.gray)
            Text("By default, your canvas will use the simulator you currently have selected (upper left in Xcode). You can preview a different device using the previewDevice modifier.")
        }
    }
}
```

```
.frame(maxWidth: .infinity)
.padding()
.background(Color.red)
.foregroundColor(.white)
}.font(.title)
```

```
}
```

```
struct Previews_Devices_Previews: PreviewProvider {
    static var previews: some View {
        Previews_Devices()
            .previewDevice("iPad Pro (9.7-inch)")
    }
}
```

How do I know what to type for a device?

Just look at your list of simulators and type in exactly as you see them displayed in the list.

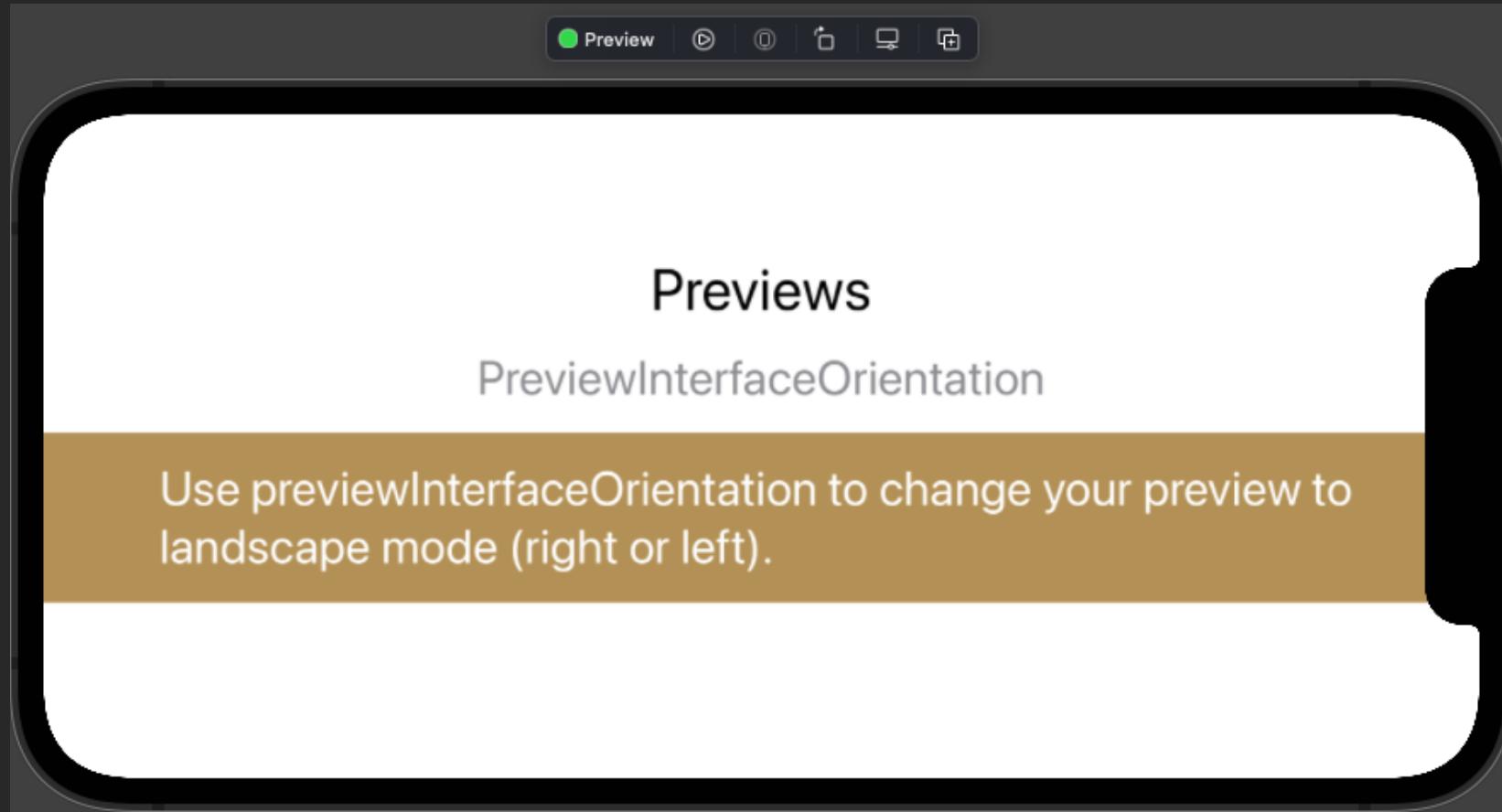
Dynamic Type Variants

While designing the UI of your app you will want to make sure it looks good for all sizes of type (which a user can change in the Settings app).

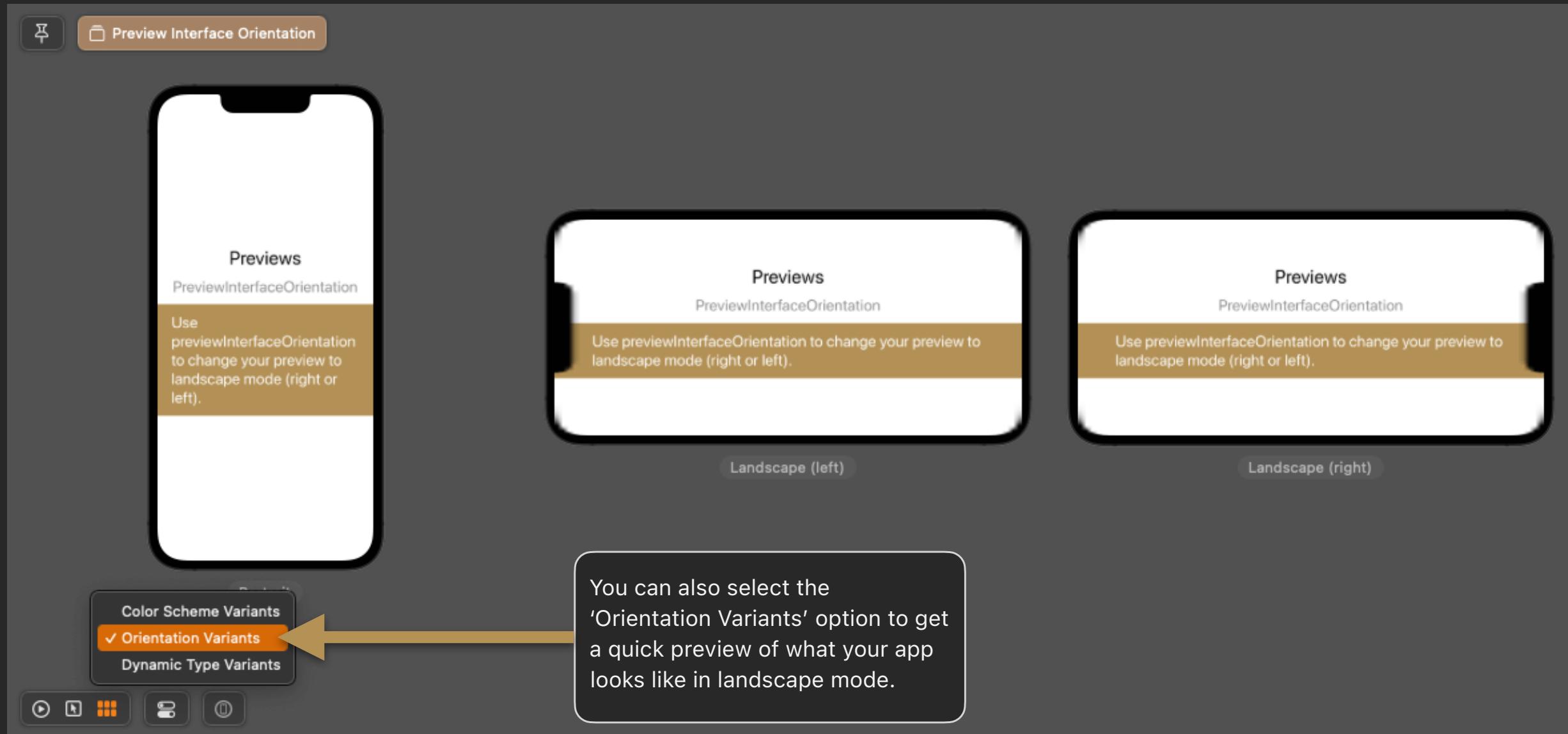
Look in the corner for preview options and select "Dynamic Type Variants".



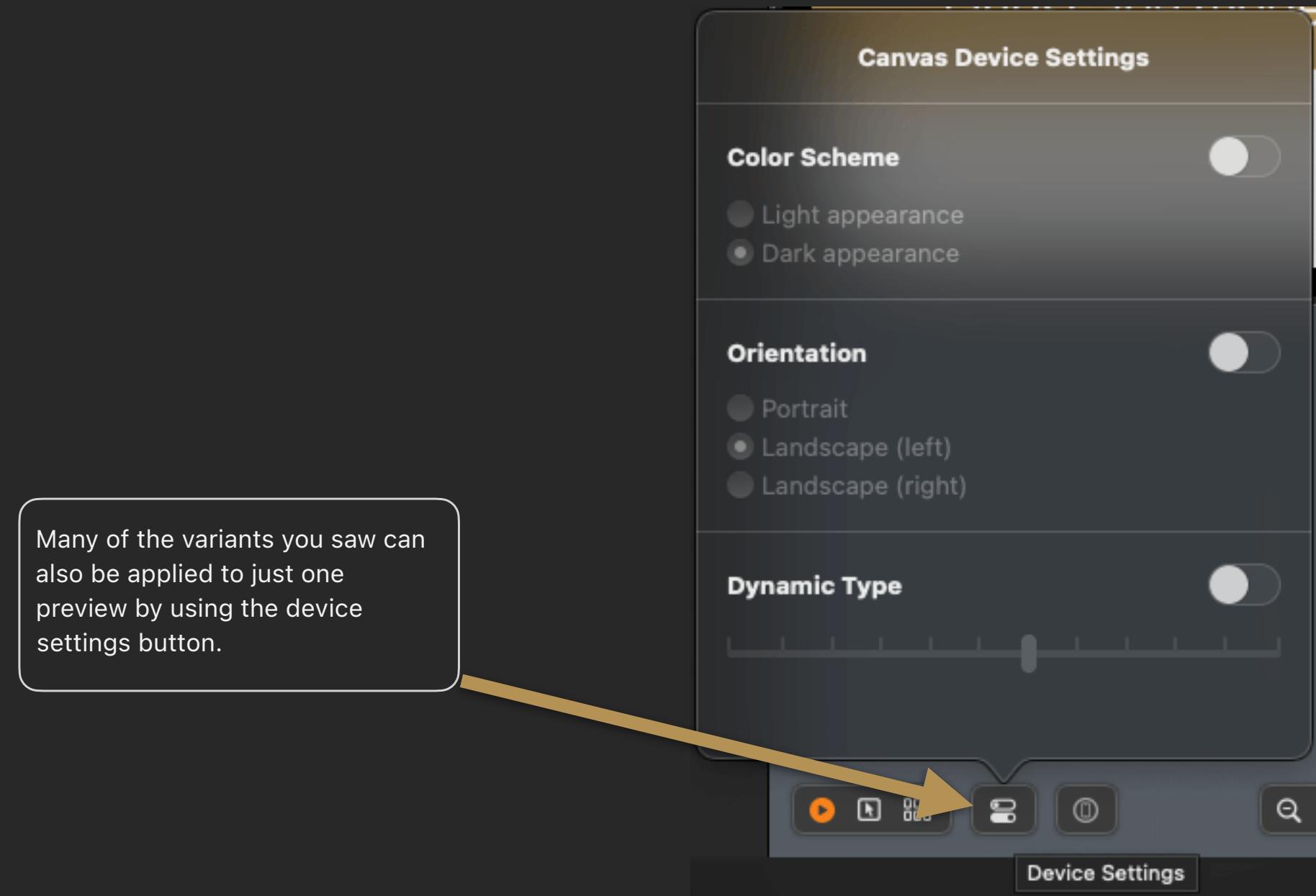
Landscape



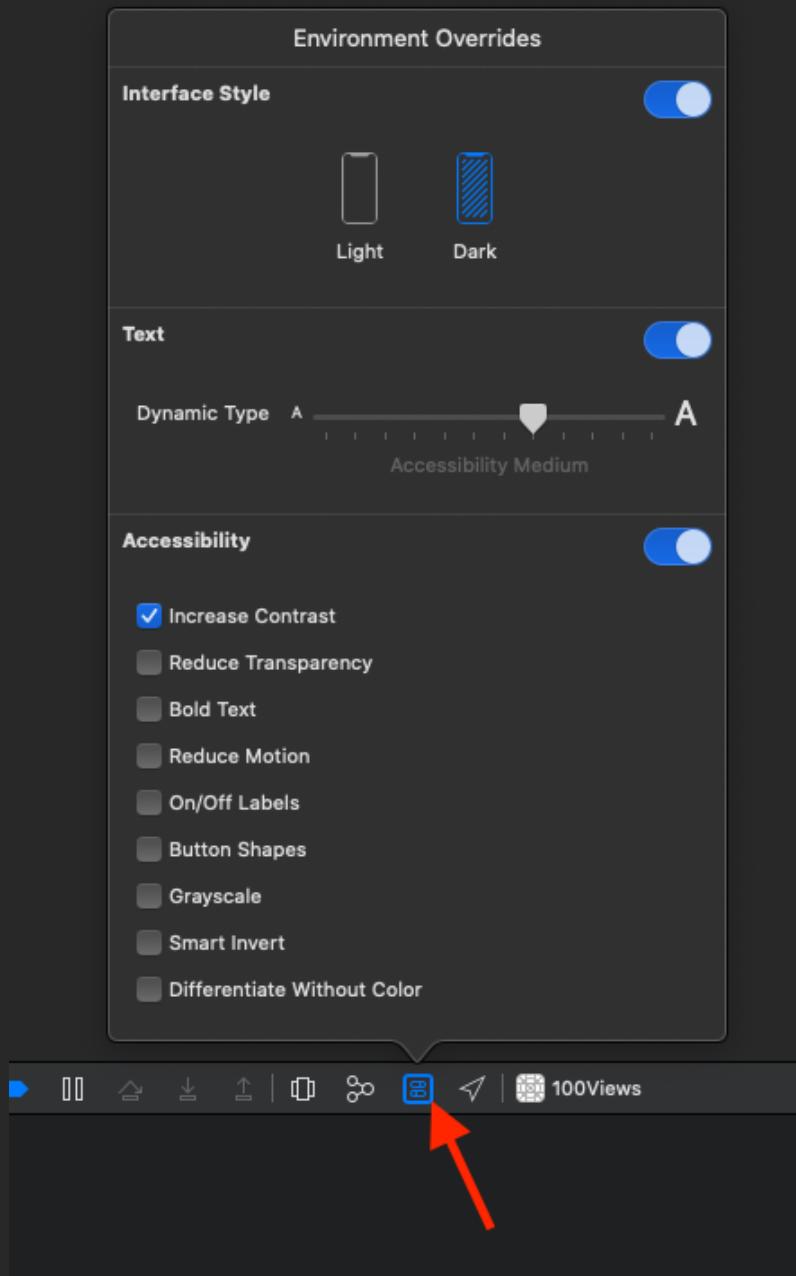
Orientation Variants



Device Settings



Environment Overrides



If you prefer to see your work in the **Simulator** then you can access many of the options mentioned through the Environment Overrides options.

This button will show up when you run your app in the debugging toolbar at the bottom of Xcode.

AYOUT VIEWS





Chapter Quick Links

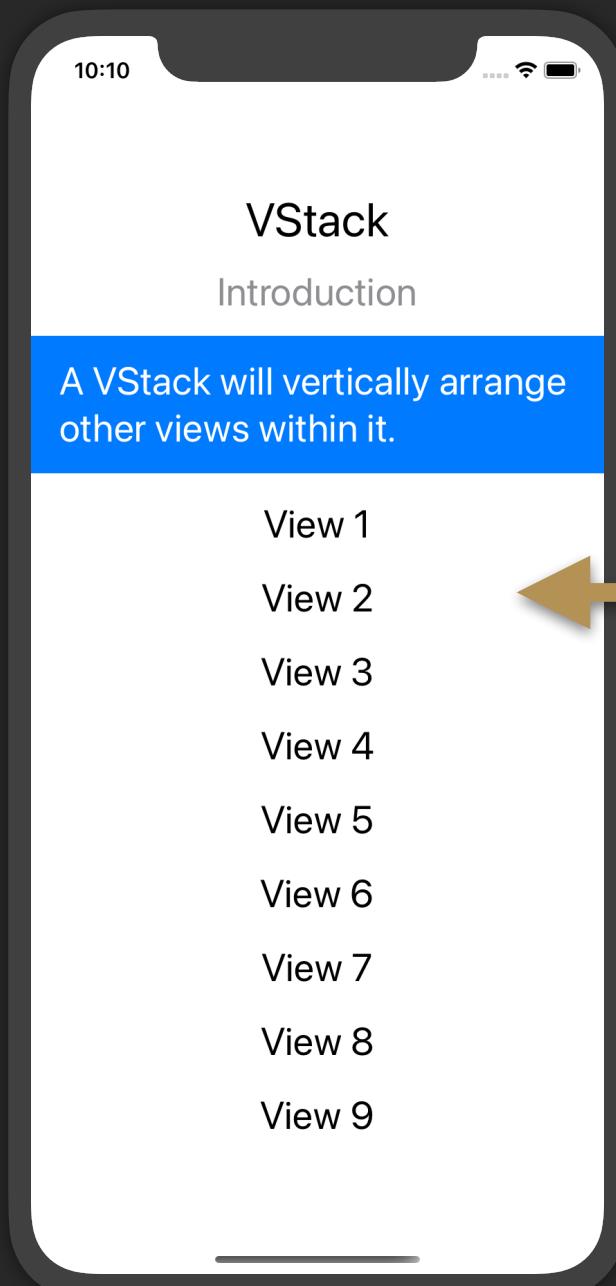
[VStack](#)[LazyVStack](#)[HStack](#)[LazyHStack](#)[Depth \(Z\) Stack](#)[Grid](#)[Spacer](#)[GeometryReader](#)[LazyHGrid](#)[LazyVGrid](#)[ScrollViewReader](#)[ControlGroup](#)[Table](#)[ViewThatFits](#)

VStack



VStack stands for “Vertical Stack”. It is a pull-in container view in which you pass in up to ten views and it will compose them one below the next, going down the screen.

Introduction



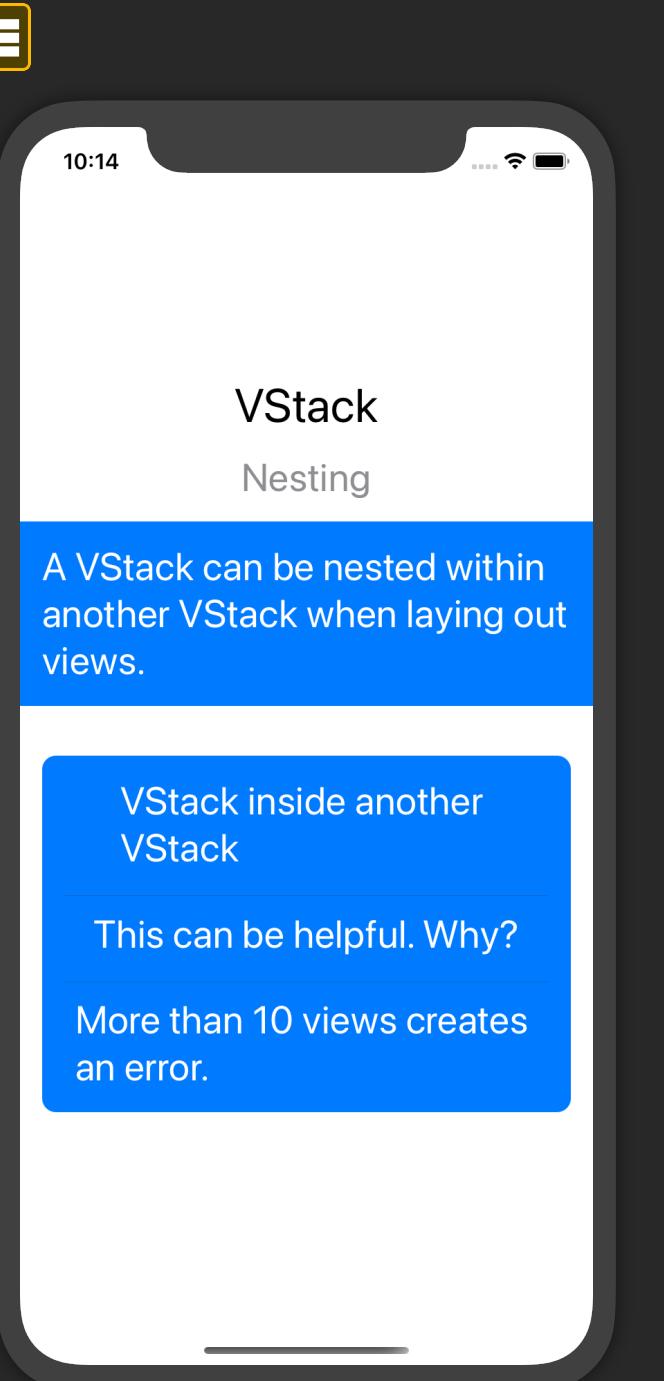
```
struct VStack_Intro : View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("VStack",  
                subtitle: "Introduction",  
                desc: "A VStack will vertically arrange other views within it.",  
                back: .blue, textColor: .white)  
  
            Text("View 1")  
            Text("View 2")  
            Text("View 3")  
            Text("View 4")  
            Text("View 5")  
            Text("View 6")  
            Text("View 7")  
            Text("View 8")  
            Text("View 9")  
        }  
        .font(.title)  
    }  
}
```

In SwiftUI, container views, like the VStack, can only contain up to 10 views.

Nesting

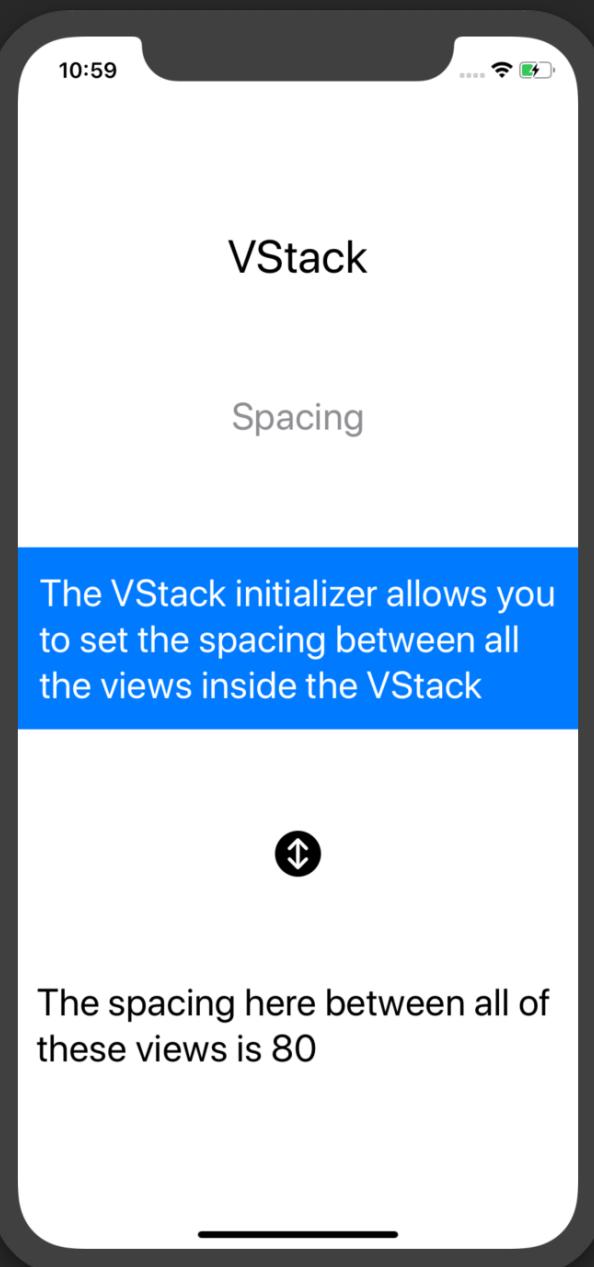
```
struct VStack_Nesting: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("VStack",  
                subtitle: "Nesting",  
                desc: "A VStack can be nested within another VStack when laying out  
                views.",  
                back: .blue, textColor: .white)  
  
            VStack {  
                Text("VStack inside another VStack")  
                Divider()  
                Text("This can be helpful. Why?")  
                Divider()  
                Text("More than 10 views creates an error.")  
            }  
            .padding()  
            .foregroundColor(.white)  
            .background(  
                // Use a blue rectangle as the background  
                RoundedRectangle(cornerRadius: 10)  
                    .foregroundColor(.blue))  
            .padding()  
            .font(.title)  
        }  
    }  
}
```

Nest container views within each other to enable you to add more views beyond the 10-view limit.



Spacing

```
VStack(spacing: 80) {  
    Text("VStack")  
        .font(.largeTitle)  
  
    Text("Spacing")  
        .font(.title)  
        .foregroundColor(.gray)  
  
    Text("The VStack initializer allows you to set the spacing between all the views inside the VStack")  
        .frame(maxWidth: .infinity)  
        .padding()  
        .background(Color.blue).font(.title)  
        .foregroundColor(.white)  
  
    Image(systemName: "arrow.up.and.down.circle.fill")  
        .font(.largeTitle)  
  
    Text("The spacing here between all of these views is 80")  
        .font(.title)  
}
```



Set spacing in the initializer.

Alignment

4:21



VStack

Alignment

By default, views in a VStack are center aligned.

Leading Alignment



Trailing Alignment



```
VStack(spacing: 20) {  
    Text("VStack")  
        .font(.largeTitle)  
    Text("Alignment")  
        .font(.title)  
        .foregroundColor(.gray)  
    Text("By default, views in a VStack are center aligned.")  
    ...  
    VStack(alignment: .leading, spacing: 40) {  
        Text("Leading Alignment")  
            .font(.title)  
        Divider() // Creates a thin line (Push-out view)  
        Image(systemName: "arrow.left")  
    }  
    .padding()  
    .foregroundColor(Color.white)  
    .background(RoundedRectangle(cornerRadius: 10)  
        .foregroundColor(.blue))  
    .padding()  
  
    VStack(alignment: .trailing, spacing: 40) {  
        Text("Trailing Alignment")  
            .font(.title)  
        Divider()  
        Image(systemName: "arrow.right")  
    }  
    .padding()  
    .foregroundColor(Color.white)  
    .background(RoundedRectangle(cornerRadius: 10)  
        .foregroundColor(.blue))  
    .padding()  
}
```

Set alignment in the initializer.



This book is a preview of:

SwiftUI Views Mastery

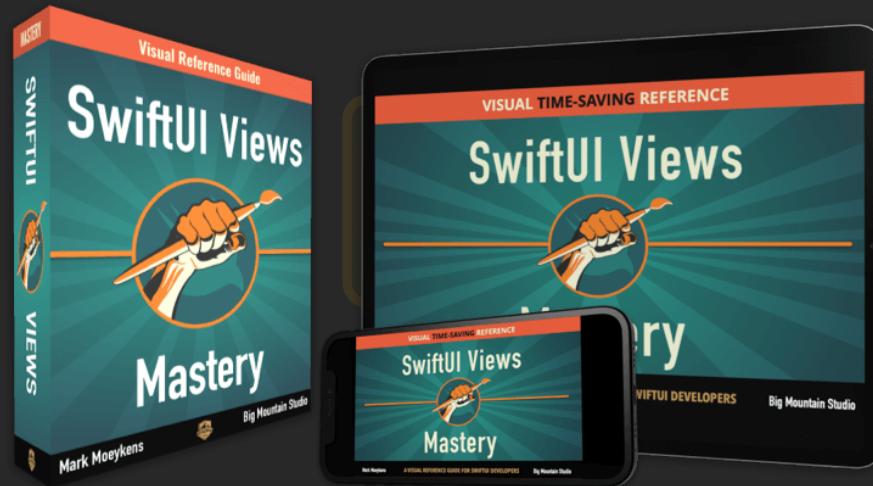


- ✓ Over **1,000** pages of SwiftUI
- ✓ Over **700** screenshots and video showing you what you can do so you can quickly come back and reference the code
- ✓ Learn all the ways to work with and modify images
- ✓ See the many ways you can use color as views
- ✓ Discover the different gradients and how you can apply them

- ✓ Find out how to implement action sheets, modals, popovers and custom popups
- ✓ Master all the layout modifiers including background and overlay layers, scaling, offsets padding and positioning
- ✓ How do you hide the status bar in SwiftUI? Find out!
- ✓ ***This is just the tip of the mountain!***

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

LazyVStack



This SwiftUI chapter is locked in this
preview.

The Lazy Vertical Stack is similar to the VStack. It's "lazy" because if you have views scrolling off the screen, SwiftUI will not load them unless it needs to show them on the screen. The VStack does not do this. The VStack loads all child views when displayed.

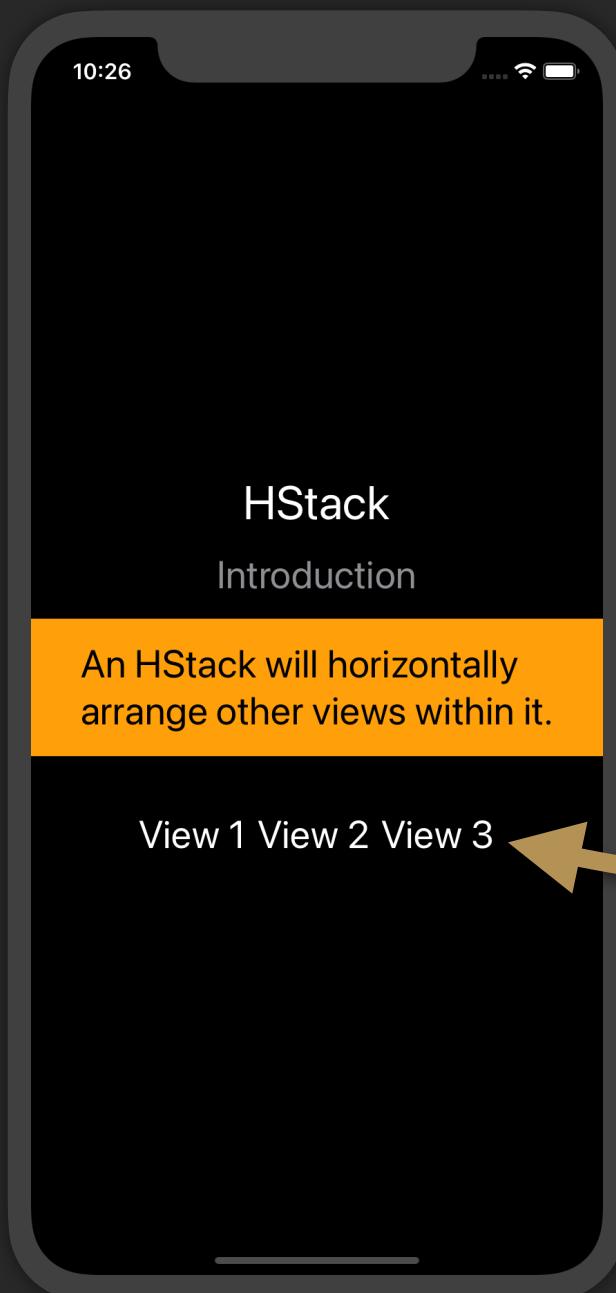
SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

HStack



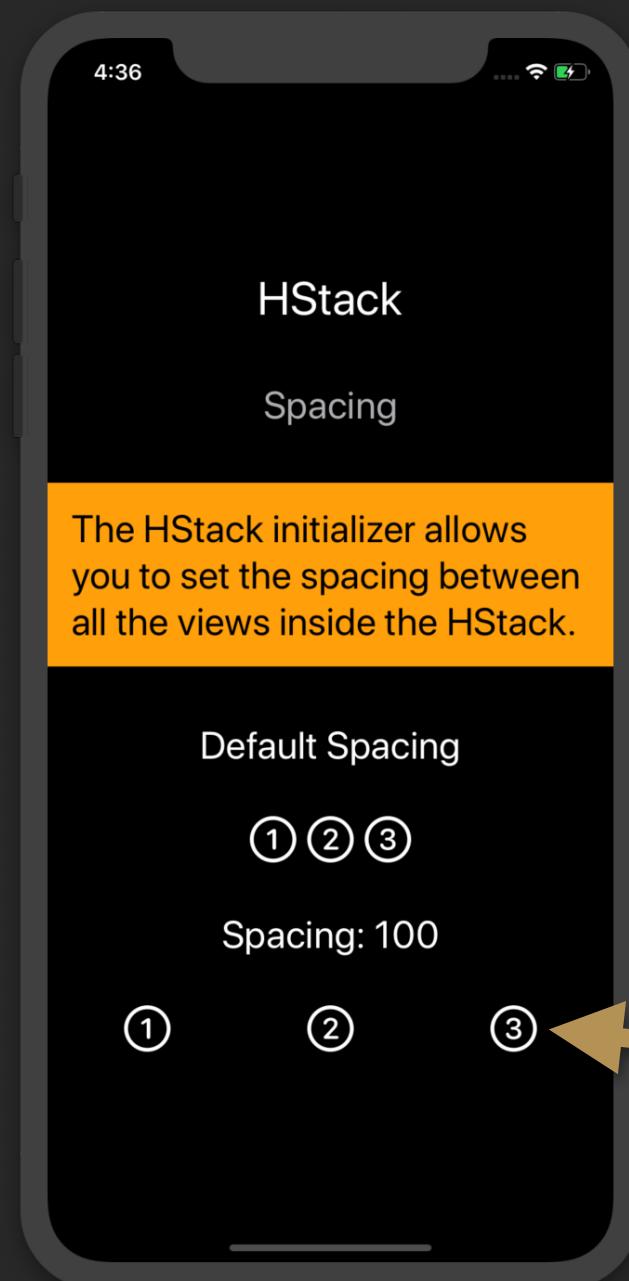
HStack stands for "Horizontal Stack". It is a pull-in container view in which you pass in up to ten views and it will compose them side-by-side.

Introduction



```
struct HStack_Intro: View {  
    var body: some View {  
        VStack(spacing: 40) {  
            HeaderView("HStack",  
                      subtitle: "Introduction",  
                      desc: "An HStack will horizontally arrange other views within it.",  
                      back: .orange)  
  
            HStack {  
                Text("View 1")  
                Text("View 2")  
                Text("View 3")  
            }  
            .font(.title)  
        }  
    }  
}
```

Spacing

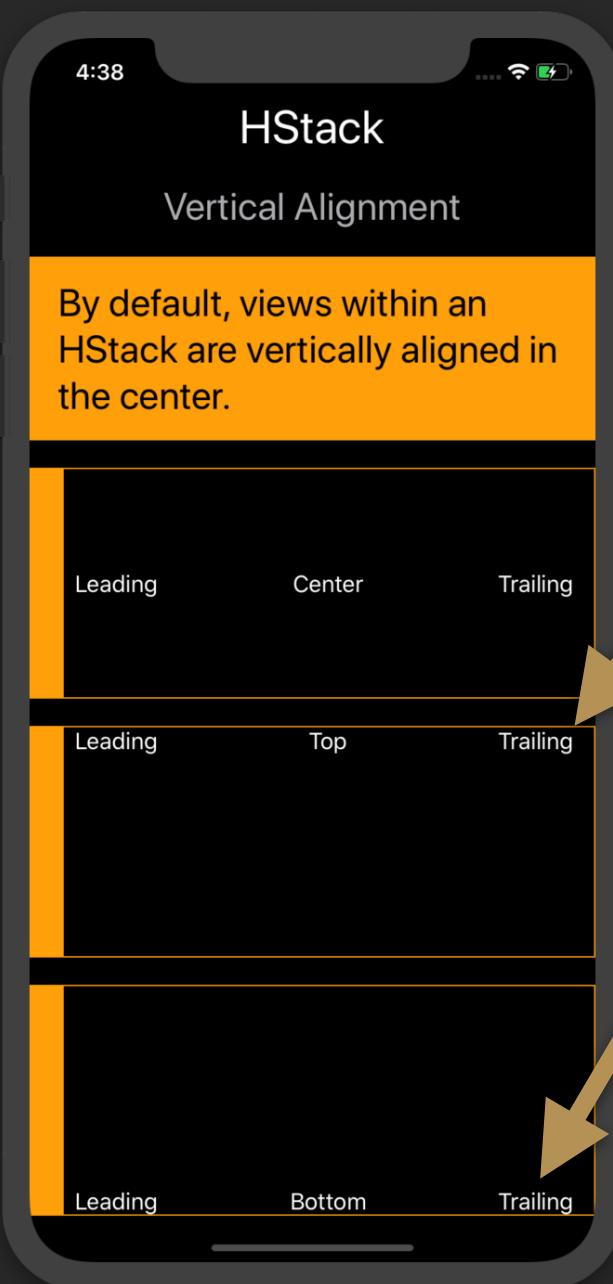


```
VStack(spacing: 40) {  
    Text("HStack")  
        .font(.largeTitle)  
  
    Text("Spacing")  
        .font(.title)  
        .foregroundColor(.gray)  
  
    Text("The HStack initializer allows you to set the spacing between all the views inside the  
        HStack")  
        .frame(maxWidth: .infinity)  
        .padding()  
        .background(Color.orange).font(.title)  
        .foregroundColor(.black)  
  
    Text("Default Spacing")  
        .font(.title)  
    HStack {  
        Image(systemName: "1.circle")  
        Image(systemName: "2.circle")  
        Image(systemName: "3.circle")  
    }.font(.largeTitle)  
  
    Divider()  
  
    Text("Spacing: 100")  
        .font(.title)  
    HStack(spacing: 100) {  
        Image(systemName: "1.circle")  
        Image(systemName: "2.circle")  
        Image(systemName: "3.circle")  
    }.font(.largeTitle)  
}
```

Set spacing in the initializer.

Alignment

4:38



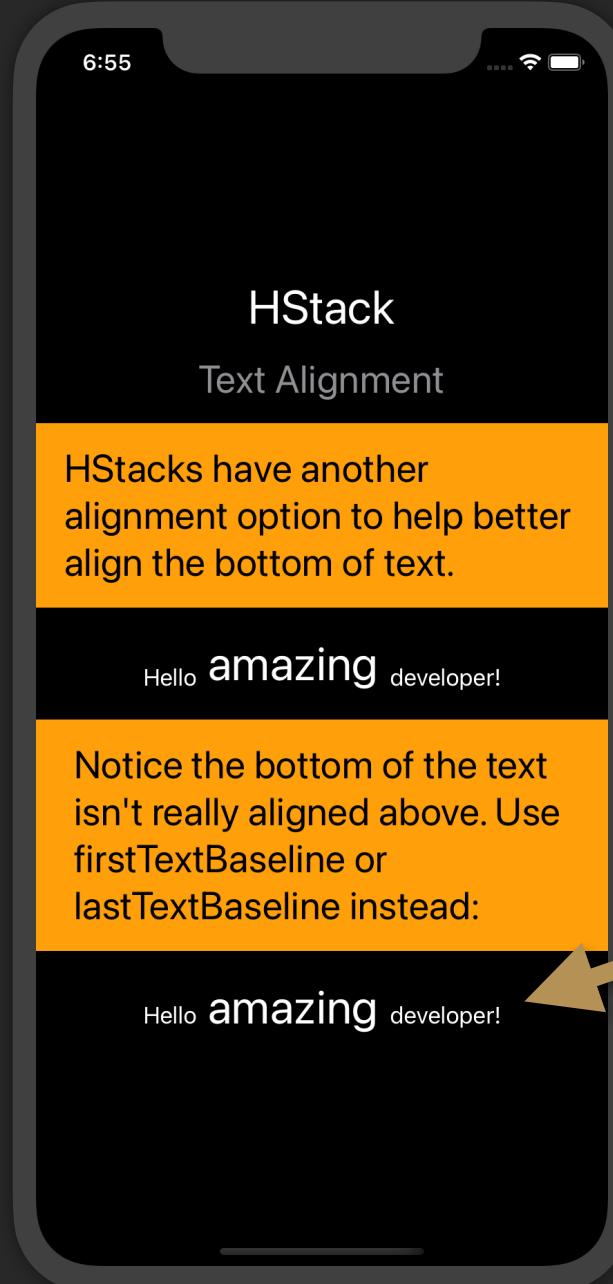
```
Text("By default, views within an HStack are vertically aligned in the center.")  
...  
HStack {  
    Rectangle().foregroundColor(.orange).frame(width: 25)  
    Text("Leading")  
    Spacer()  
    Text("Center")  
    Spacer()  
    Text("Trailing")  
        .padding(.trailing)  
}  
.border(Color.orange)  
HStack(alignment: .top) {  
    Rectangle().foregroundColor(.orange).frame(width: 25)  
    Text("Leading")  
    Spacer()  
    Text("Top")  
    Spacer()  
    Text("Trailing")  
        .padding(.trailing)  
}  
.border(Color.orange)  
HStack(alignment: .bottom) {  
    Rectangle().foregroundColor(.orange).frame(width: 25)  
    Text("Leading")  
    Spacer()  
    Text("Bottom")  
    Spacer()  
    Text("Trailing")  
        .padding(.trailing)  
}  
.border(Color.orange)
```

Set alignment in the initializer.

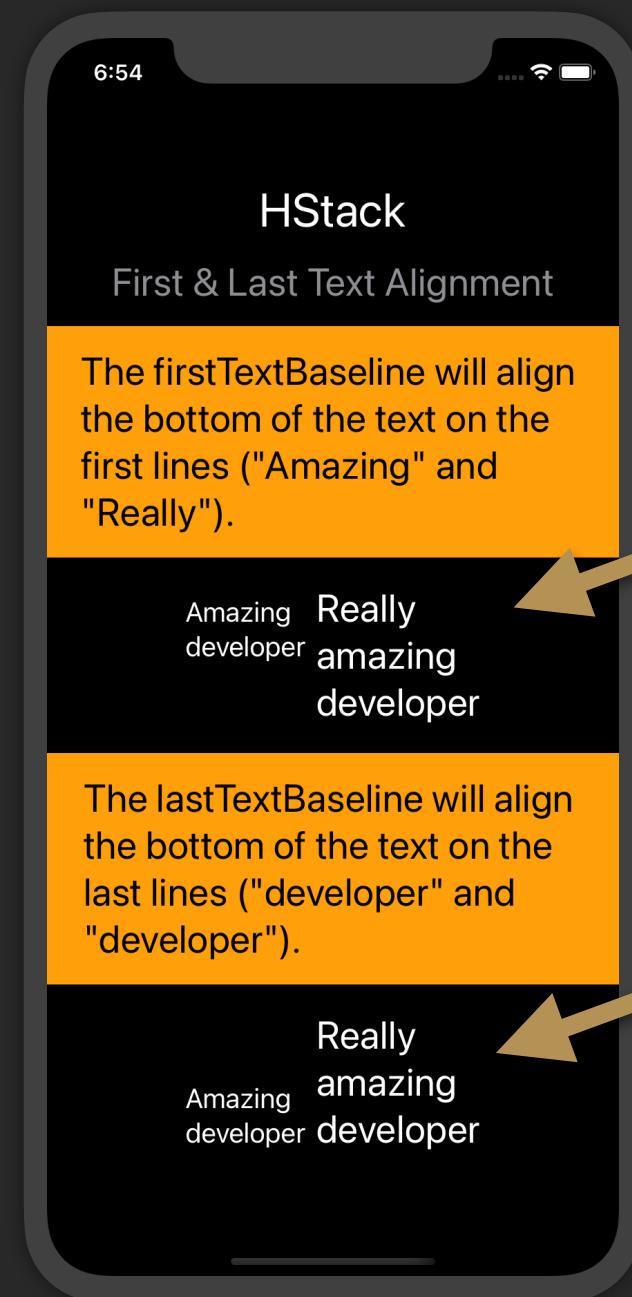
Text Alignment

```
struct HStack_TextAlignment: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("HStack",  
                subtitle: "Text Alignment",  
                desc: "HStacks have another alignment option to help better align the  
                    bottom of text.",  
                back: .orange)  
  
            HStack(alignment: .bottom) {  
                Text("Hello")  
                Text("amazing")  
                    .font(.largeTitle)  
                Text("developer!")  
            }  
            .font(.body)  
  
            DescView(desc: "Notice the bottom of the text isn't really aligned above. Use  
                firstTextBaseline or lastTextBaseline instead:", back: .orange)  
  
            HStack(alignment: .firstTextBaseline) {  
                Text("Hello")  
                Text("amazing")  
                    .font(.largeTitle)  
                Text("developer!")  
            }  
            .font(.body)  
            .font(.title)  
        }  
    }  
}
```

This will align the text normally.
But what's the difference between first and
last text baseline? See on the next page.



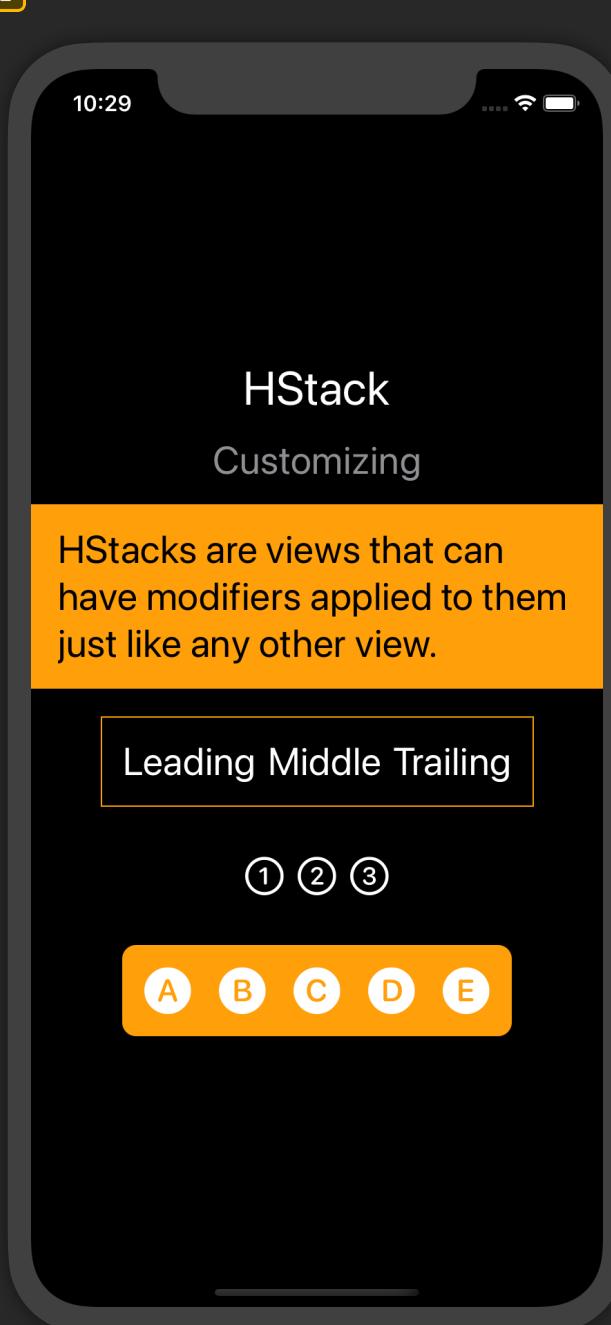
First & Last Text Alignment



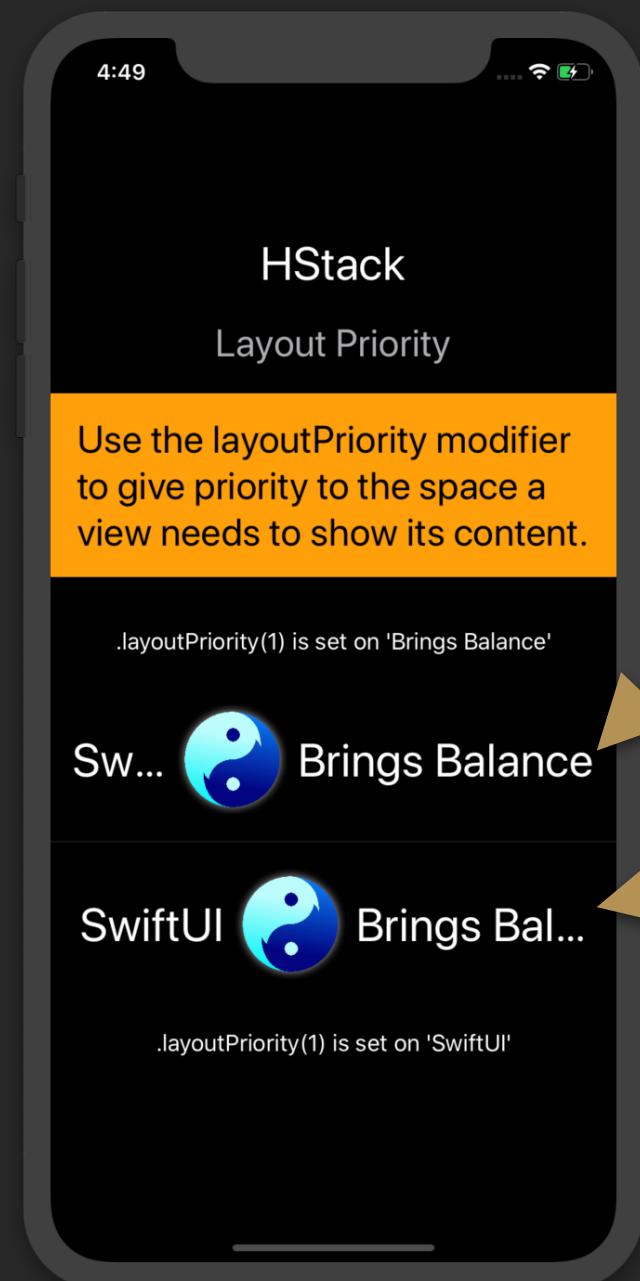
```
struct HStack_TextAlignment_FirstLast: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("HStack",  
                subtitle: "First & Last Text Alignment",  
                desc: "The firstTextBaseline will align the bottom of the text on the  
                    first lines (\\"Amazing\\" and \\"Really\\").",  
                back: .orange)  
            HStack(alignment: .firstTextBaseline) {  
                Text("Amazing developer")  
                    .font(.title3)  
                Text("Really amazing developer")  
            }  
            .frame(width: 250)  
  
            DescView(desc: "The lastTextBaseline will align the bottom of the text on the last  
                lines (\\"developer\\" and \\"developer\\").", back: .orange)  
            HStack(alignment: .lastTextBaseline) {  
                Text("Amazing developer")  
                    .font(.title3)  
                Text("Really amazing developer")  
            }  
            .frame(width: 250)  
        }  
        .font(.title)  
    }  
}
```

Customization

```
struct HStack_Customizing : View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("HStack",  
                subtitle: "Customizing",  
                desc: "HStacks are views that can have modifiers applied to them just  
like any other view.",  
                back: .orange)  
  
            HStack {  
                Text("Leading")  
                Text("Middle")  
                Text("Trailing")  
            }  
            .padding()  
            .border(Color.orange) // Create a 2 point border using the color specified  
  
            HStack(spacing: 10) {  
                Image(systemName: "1.circle")  
                Image(systemName: "2.circle")  
                Image(systemName: "3.circle")  
            }.padding()  
  
            HStack(spacing: 20) {  
                Image(systemName: "a.circle.fill")  
                Image(systemName: "b.circle.fill")  
                Image(systemName: "c.circle.fill")  
                Image(systemName: "d.circle.fill")  
                Image(systemName: "e.circle.fill")  
            }  
            .font(.largeTitle).padding()  
            .background(RoundedRectangle(cornerRadius: 10)  
            .foregroundColor(.orange))  
        }  
        .font(.title)  
    }  
}
```



Layout Priority



When using a horizontal stack with text views within it, there's a chance that text might truncate if you are not allowing them to wrap. In this case, you can prioritize which one will truncate last with layout priority. The default value is 0. The higher the number, the higher the priority to have enough space to not be truncated.

```
HStack {  
    Text("SwiftUI")  
        .font(.largeTitle).lineLimit(1) // Don't let text wrap  
    Image("SwiftUI")  
        .resizable()  
        .frame(width: 80, height: 80)  
    Text("Brings Balance")  
        .font(.largeTitle)  
        .layoutPriority(1) // Truncate last ←  
    }  
    .padding([.horizontal])  
    Divider()
```

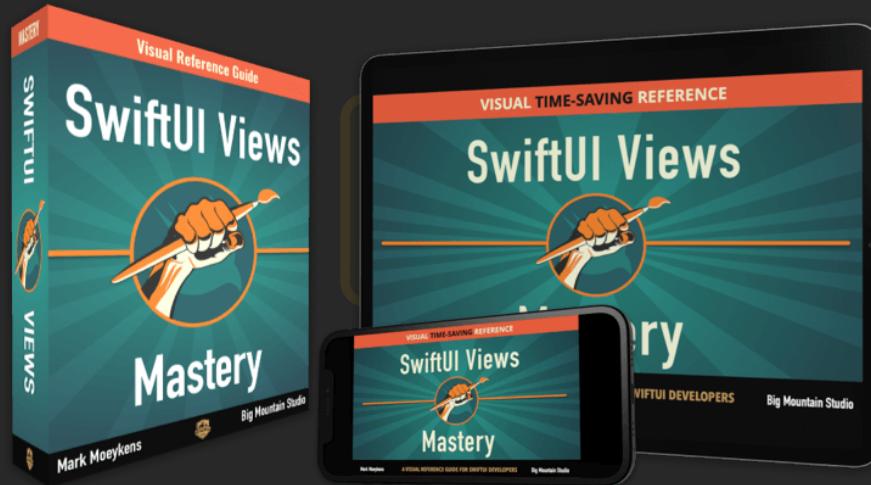


```
HStack {  
    Text("SwiftUI")  
        .font(.largeTitle)  
        .layoutPriority(1) // Truncate last ←  
    Image("SwiftUI")  
        .resizable()  
        .frame(width: 80, height: 80)  
    Text("Brings Balance")  
        .font(.largeTitle).lineLimit(1) // Don't let text wrap  
    }  
    .padding(.horizontal)
```



Note: You can learn more about layout priority in the chapter "[Layout Modifiers](#)", section "LayoutPriority".

LazyHStack



This SwiftUI chapter is locked in this
preview.

The Lazy Horizontal Stack is similar to the HStack. It's "lazy" because if you have views scrolling off the screen, SwiftUI will not load them unless it needs to show them on the screen. The HStack does not do this. The HStack loads all child views when displayed.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Depth (Z) Stack

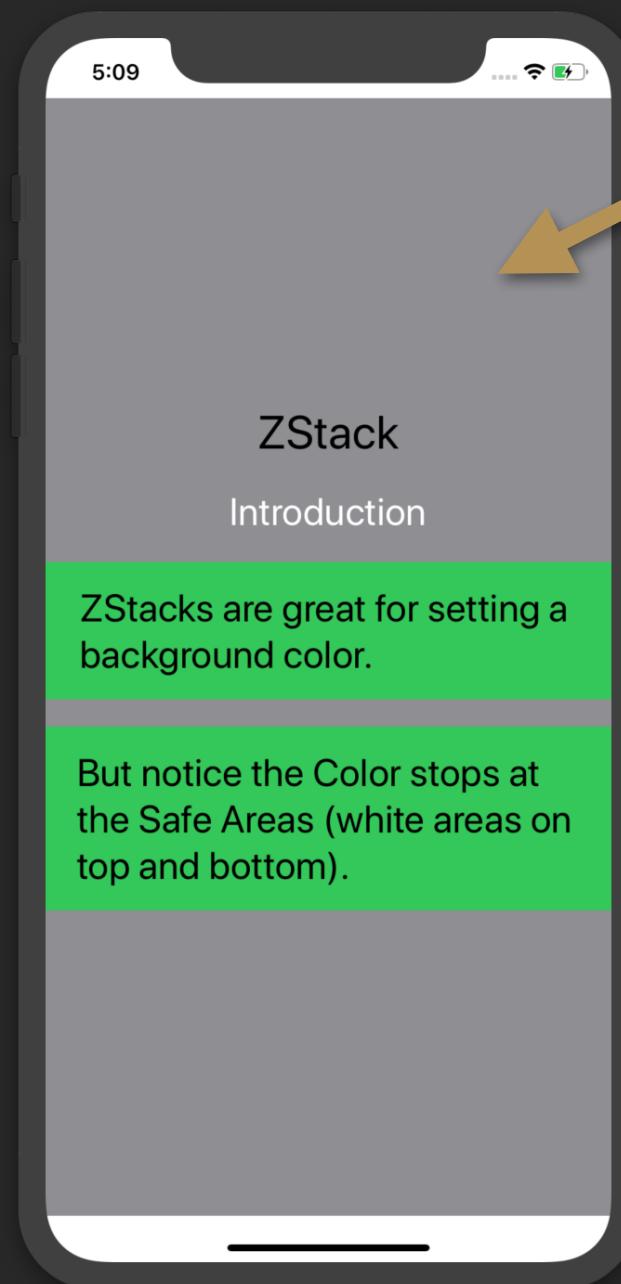


A Depth Stack (ZStack) is a pull-in container view. It is a view that overlays its child views on top of each other. ("Z" represents the Z-axis which is depth-based in a 3D space.)

You learned earlier about creating layers with the background and overlay modifiers. ZStack is another way to create layers with views that control their own sizing and spacing.

So, the ZStack is a pull-in container view but you may think it is a push-out view because of the first example but it's actually the color that is pushing out.

Introduction

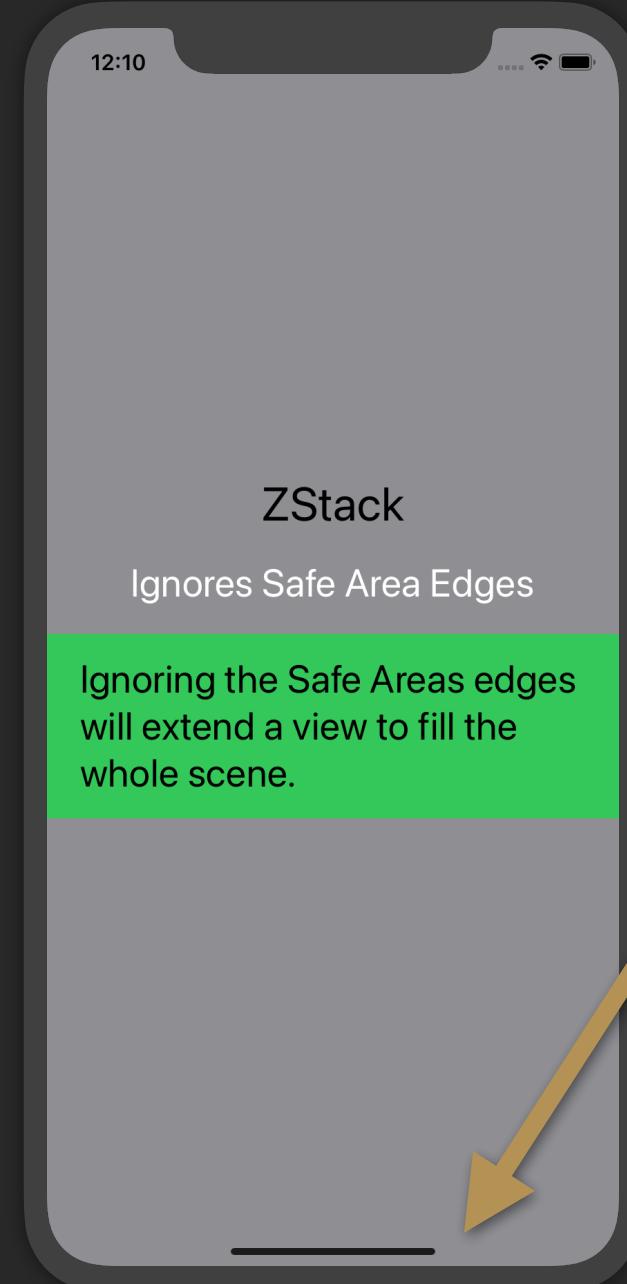


```
ZStack {  
    // LAYER 1: Furthest back  
    Color.gray // Yes, Color is a view!  
  
    // LAYER 2: This VStack is on top.  
    VStack(spacing: 20) {  
        Text("ZStack")  
            .font(.largeTitle)  
  
        Text("Introduction")  
            .foregroundColor(.white)  
  
        Text("ZStacks are great for setting a background color.")  
            .frame(maxWidth: .infinity)  
            .padding()  
            .background(Color.green)  
  
        Text("But notice the Color stops at the Safe Areas (white areas on top and bottom).")  
            .frame(maxWidth: .infinity)  
            .padding()  
            .background(Color.green)  
    }  
    .font(.title)  
}
```

You set depth by the order of the views inside the ZStack.

Note: The Color view is a push-out view. It is pushing out the ZStack container view.

Ignores Safe Area Edges



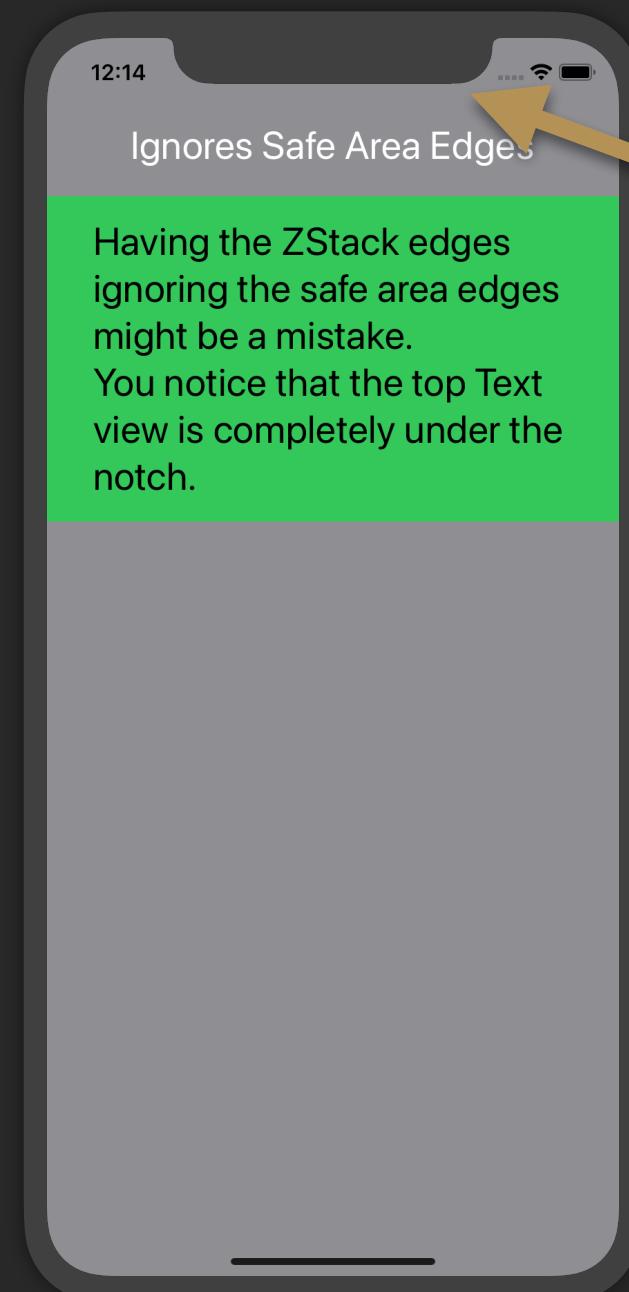
```
ZStack {  
    Color.gray  
  
    VStack(spacing: 20) {  
        Text("ZStack")  
            .font(.largeTitle)  
  
        Text("Edges Ignoring Safe Area")  
            .foregroundColor(.white)  
  
        Text("Ignoring the Safe Areas will extend a view to fill the whole scene.")  
            .frame(maxWidth: .infinity)  
            .padding()  
            .foregroundColor(.white)  
            .background(Color.green)  
  
    }  
    .font(.title)  
}  
.ignoresSafeArea(.all) // Ignore the safe areas
```



Allows views to extend past the safe areas.

Learn more about what Safe Areas are and ways to ignore edges in the chapter "[Layout Modifiers](#)" in the section "Ignores Safe Area".

Background Problem



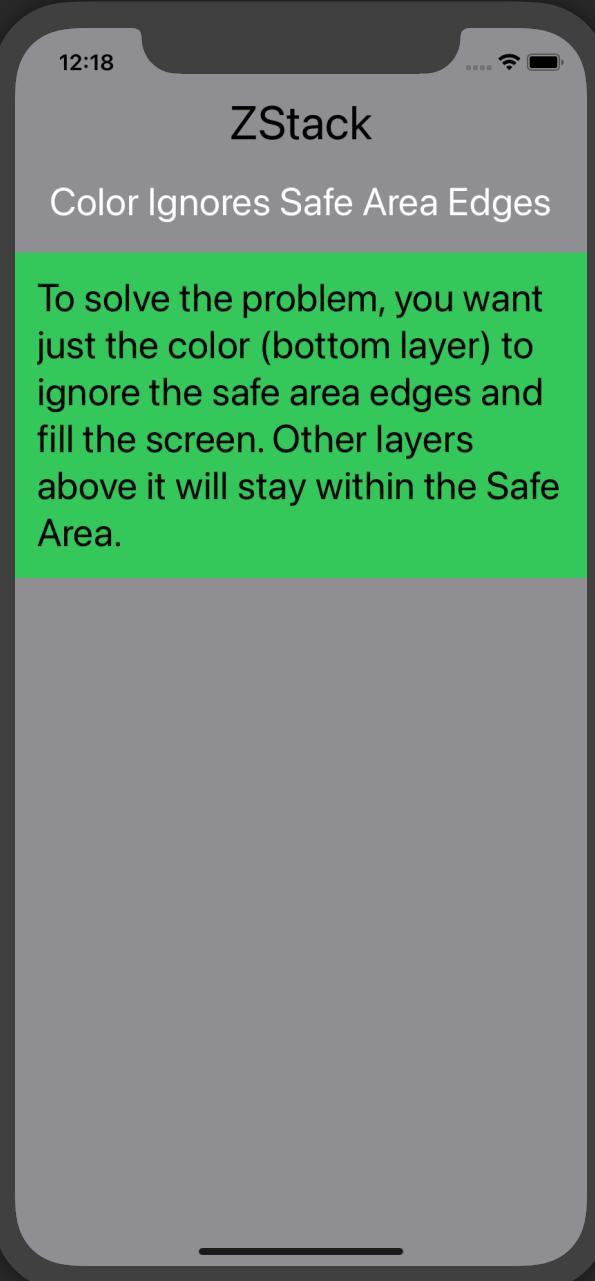
```
struct ZStack_BackgroundColor_Problem: View {  
    var body: some View {  
        ZStack {  
            Color.gray  
  
            VStack(spacing: 20) {  
                Text("ZStack") // This view is under the notch  
                    .font(.largeTitle)  
  
                Text("Ignores Safe Area Edges")  
                    .foregroundColor(.white)  
  
                Text("Having the ZStack edges ignoring the safe area edges might be a mistake.  
                    \nYou notice that the top Text view is completely under the notch.")  
                    .frame(maxWidth: .infinity)  
                    .padding()  
                    .background(Color.green)  
  
                Spacer() // Added a spacer to push the views up.  
            }  
            .font(.title)  
        }  
        .ignoresSafeArea()  
    }  
}
```

Ignores all Safe Area edges.

Background Solution

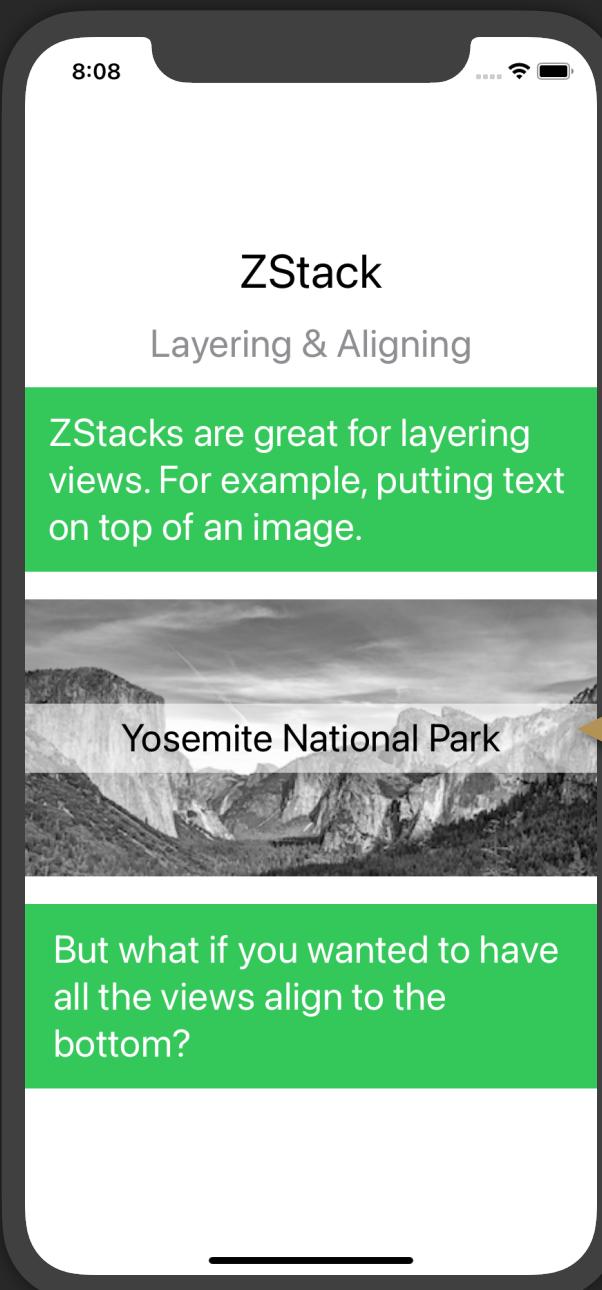
```
struct ZStack_BackgroundColor_Solution: View {  
    var body: some View {  
        ZStack {  
            Color.gray  
                .ignoresSafeArea() // Have JUST the color ignore the safe areas edges, not  
the VStack.  
                VStack(spacing: 20) {  
                    Text("ZStack")  
                        .font(.largeTitle)  
  
                    Text("Color Ignores Safe Area Edges")  
                        .foregroundColor(.white)  
  
                    Text("To solve the problem, you want just the color (bottom layer) to ignore  
the safe area edges and fill the screen. Other layers above it will stay  
within the Safe Area.")  
                        .frame(maxWidth: .infinity)  
                        .padding()  
                        .background(Color.green)  
                    Spacer()  
                }  
                .font(.title)  
            }  
        }  
    }  
}
```

Remove ignoresSafeArea modifier from the ZStack and put it just on the color.



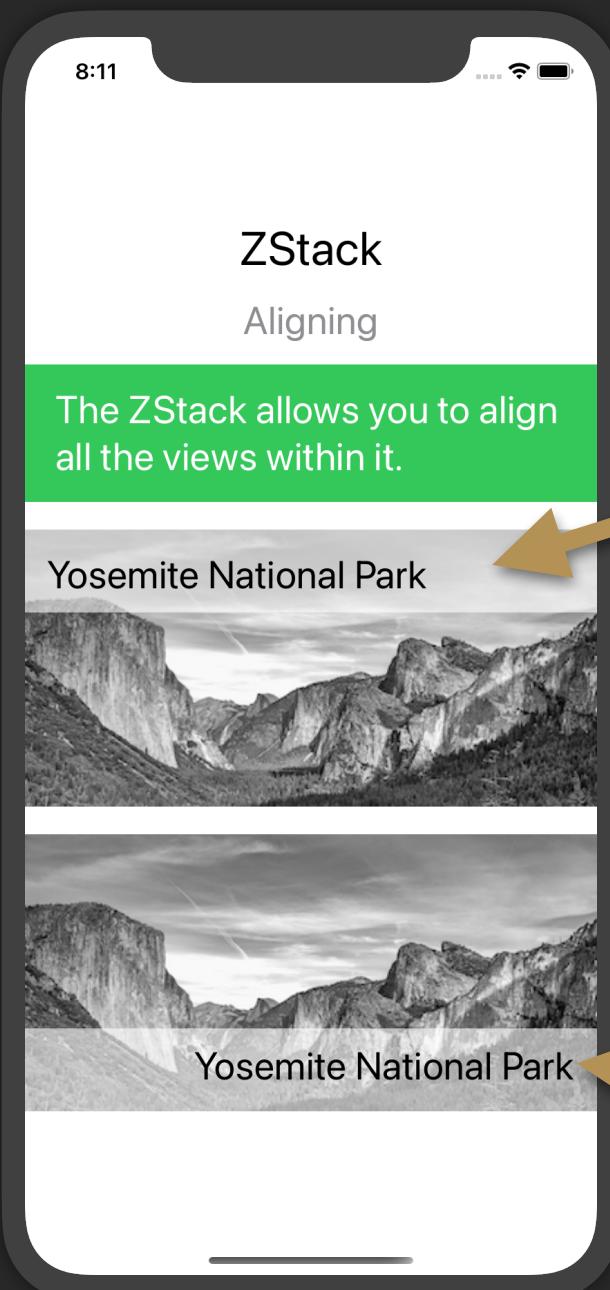
Layering

```
struct ZStack_Layering: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("ZStack",  
                subtitle: "Layering & Aligning",  
                desc: "ZStacks are great for layering views. For example, putting text on  
                top of an image.", back: .green, textColor: .white)  
            ZStack {  
                Image("yosemite_large")  
                    .resizable() // Allows image to change size  
                    .scaledToFit() // Keeps image the same aspect ratio when resizing  
  
                Rectangle()  
                    .fill(Color.white.opacity(0.6))  
                    .frame(maxWidth: .infinity, maxHeight: 50)  
  
                Text("Yosemite National Park")  
                    .font(.title)  
                    .padding()  
            }  
            DescView(desc: "But what if you wanted to have all the views align to the bottom?",  
                back: .green, textColor: .white)  
        }  
        .font(.title)  
    }  
}
```



Aligning

```
struct ZStack_Aligning: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("ZStack",  
                subtitle: "Aligning",  
                desc: "The ZStack allows you to align all the views within it.",  
                back: .green, textColor: .white)  
            ZStack(alignment: .topLeading) {  
                Image("yosemite_large")  
                    .resizable()  
                    .aspectRatio(contentMode: .fit)  
                Rectangle()  
                    .fill(Color.white.opacity(0.6))  
                    .frame(maxWidth: .infinity, maxHeight: 60)  
                Text("Yosemite National Park")  
                    .font(.title)  
                    .padding()  
            }  
            ZStack(alignment: .bottomTrailing) {  
                Image("yosemite_large")  
                    .resizable()  
                    .aspectRatio(contentMode: .fit)  
                Rectangle()  
                    .fill(Color.white.opacity(0.6))  
                    .frame(maxWidth: .infinity, maxHeight: 60)  
                Text("Yosemite National Park")  
                    .font(.title)  
                    .padding()  
            }  
        }  
    }  
}
```



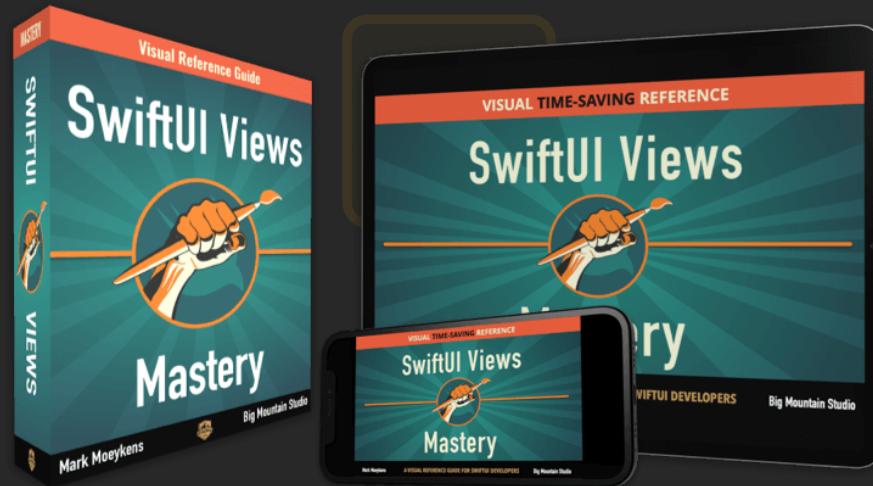
Use the alignment parameter in the ZStack's initializer to set where you want all views within to be aligned.



Alignment Choices

- center
 - leading
 - trailing
 - top
 - bottom
 - topLeading
 - topTrailing
 - bottomLeading
 - bottomTrailing

Grid



This SwiftUI chapter is locked in this
preview.

The grid will layout views in rows and columns. What is different about the grid and other layout views is that it will remember and maintain the widths of all the views in each row and apply the same width to all views in that same column.

Grids are pull-in views, they

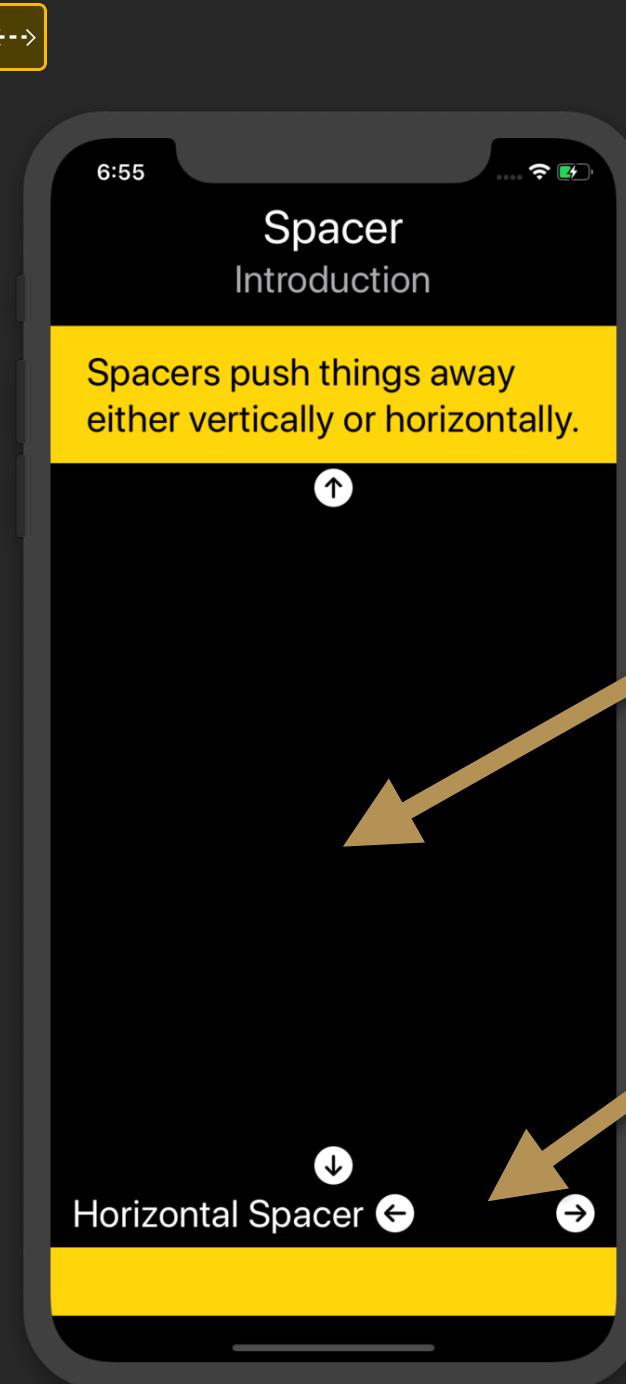
SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Spacer



You may notice that when you add new pull-in views, such as Text views, they appear in the center of the screen. You can use the Spacer to push these views apart, away from the center of the screen.

Introduction

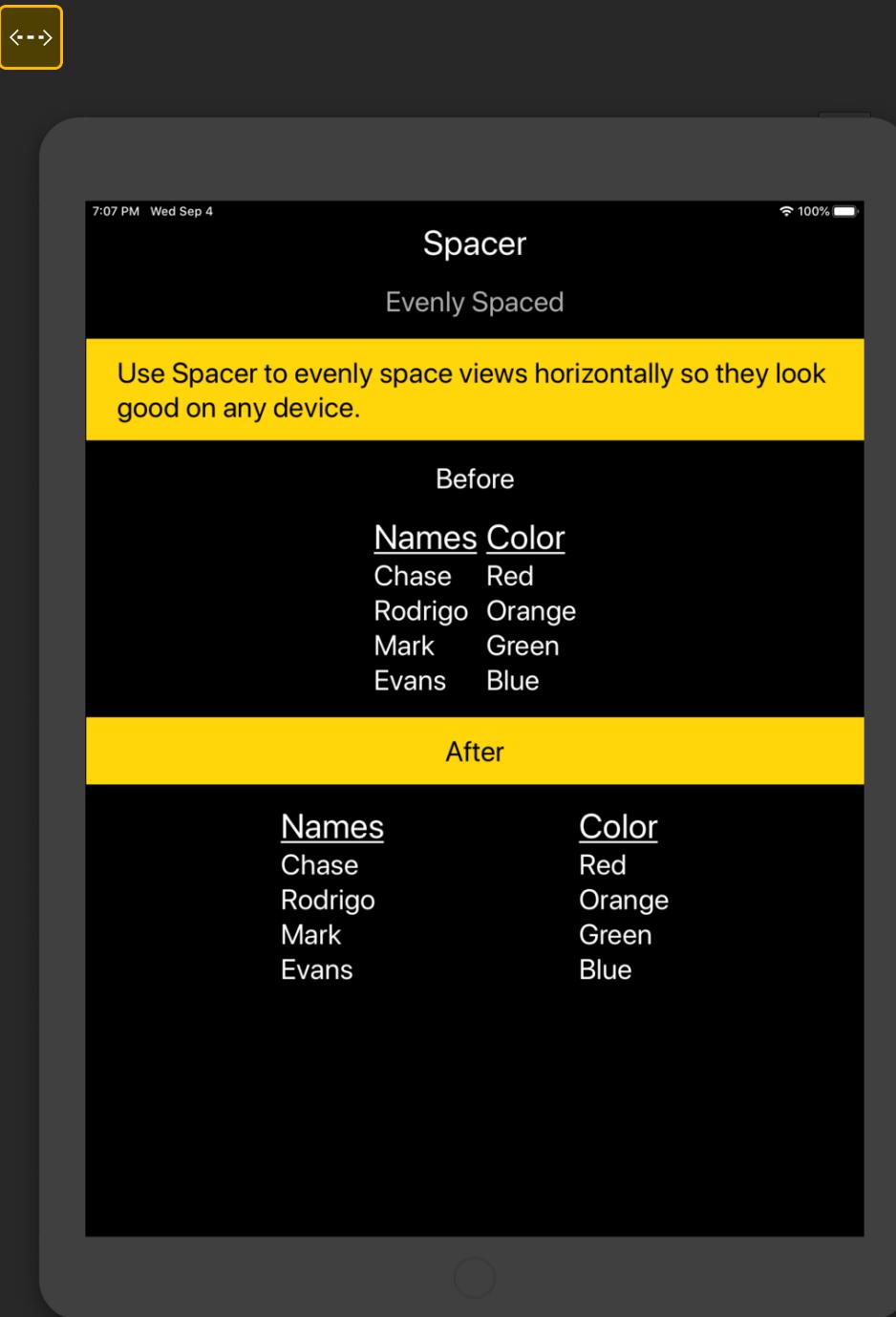


```
Spacer {  
    Text("Spacer")  
        .font(.largeTitle)  
  
    Text("Introduction")  
        .foregroundColor(.gray)  
  
    Text("Spacers push things away either vertically or horizontally")  
        ...  
  
    Image(systemName: "arrow.up.circle.fill")  
    Spacer()  
    Image(systemName: "arrow.down.circle.fill")  
  
    HStack {  
        Text("Horizontal Spacer")  
  
        Image(systemName: "arrow.left.circle.fill")  
        Spacer()  
        Image(systemName: "arrow.right.circle.fill")  
    }  
    .padding(.horizontal)  
  
    Color.yellow  
        .frame(maxHeight: 50) // Height can decrease but not go higher than 50  
    }  
    .font(.title) // Apply this font to every view within the VStack
```

Pushes away vertically when in a VStack.

Pushes away horizontally when in an HStack.

Evenly Spaced



```
Text("Use Spacer to evenly space views horizontally so they look good on any device.")
```

```
Text("After")
```

```
HStack {
```

```
    Spacer()
```

```
    VStack(alignment: .leading) {
```

```
        Text("Names")
```

```
        .font(.largeTitle)  
        .underline()
```

```
        Text("Chase")
```

```
        Text("Rodrigo")
```

```
        Text("Mark")
```

```
        Text("Evans")
```

```
}
```

```
    Spacer()
```

```
    VStack(alignment: .leading) {
```

```
        Text("Color")
```

```
        .font(.largeTitle)  
        .underline()
```

```
        Text("Red")
```

```
        Text("Orange")
```

```
        Text("Green")
```

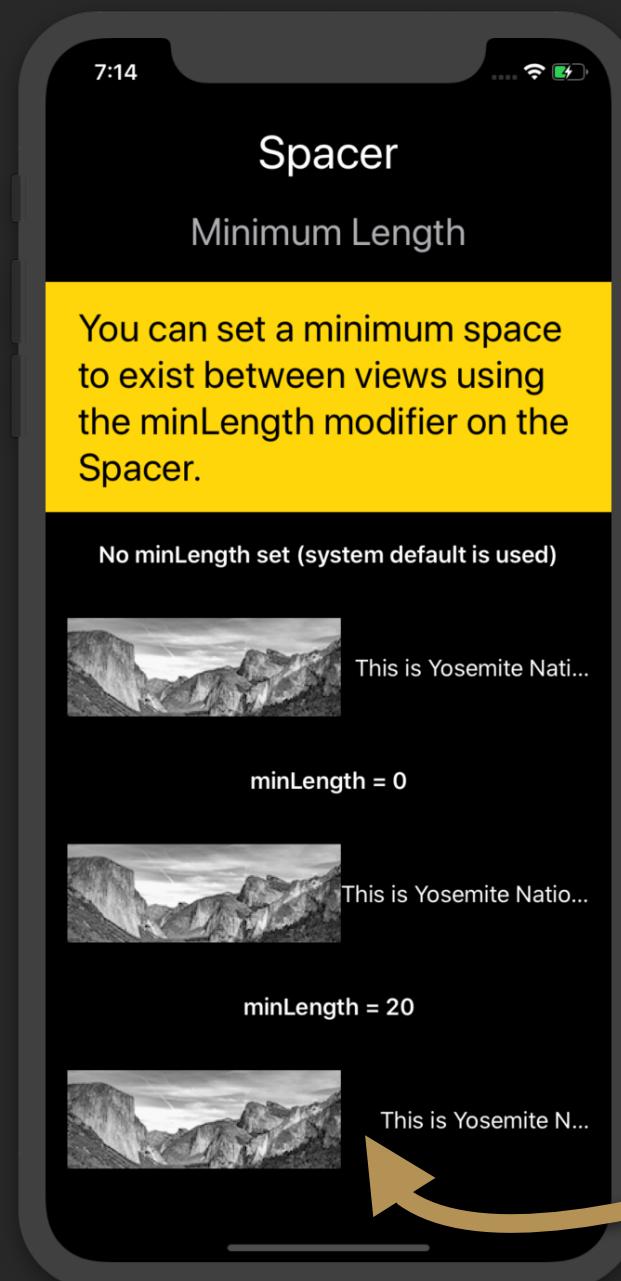
```
        Text("Blue")
```

```
}
```

```
    Spacer()
```

```
}
```

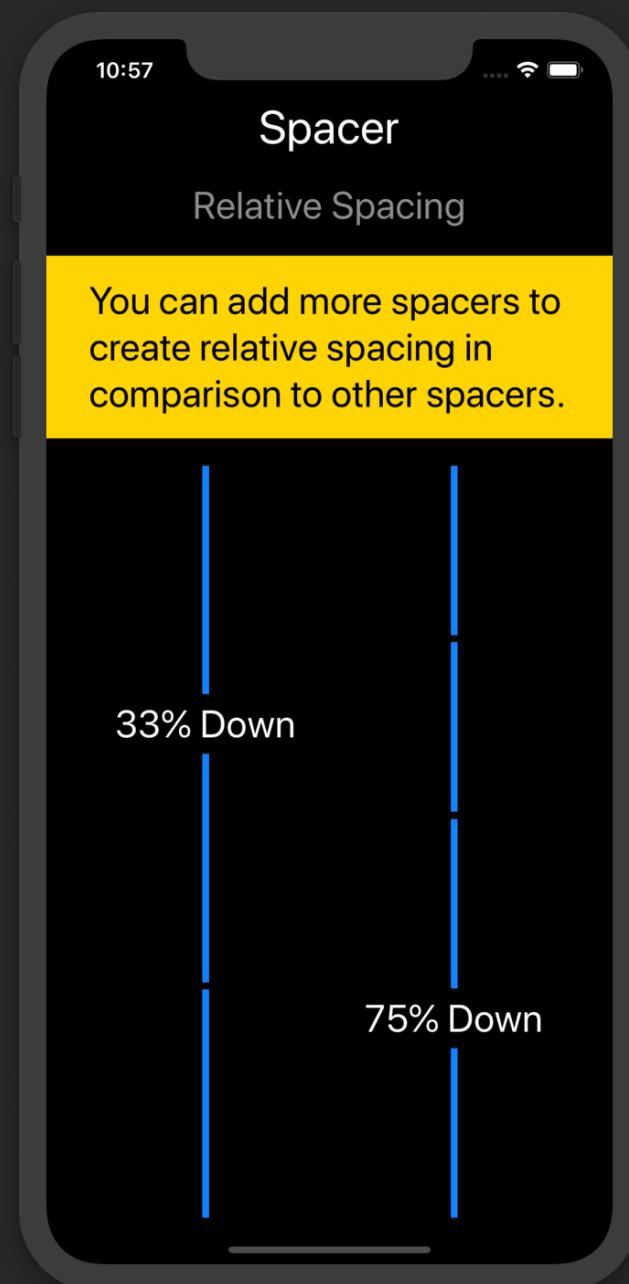
Minimum Length



```
VStack(spacing: 10) {  
    Text("Spacer")  
        .font(.largeTitle)  
    Text("Minimum Length")  
        .font(.title)  
        .foregroundColor(.gray)  
    Text("You can set a minimum space to exist between views using the minLength modifier on the  
        Spacer.")  
        ...  
    Text("No minLength set (system default is used)")  
        .bold()  
    HStack {  
        Image("yosemite")  
        Spacer()  
        Text("This is Yosemite National Park").lineLimit(1)  
    }.padding()  
  
    Text("minLength = 0")  
        .bold()  
    HStack {  
        Image("yosemite")  
        Spacer(minLength: 0)  
        Text("This is Yosemite National Park").lineLimit(1)  
    }.padding()  
  
    Text("minLength = 20")  
        .bold()  
    HStack {  
        Image("yosemite")  
        Spacer(minLength: 20)  
        Text("This is Yosemite National Park").lineLimit(1)  
    }.padding()  
}
```

Set the minimum length in the Spacer's initializer.

Relative Spacing with Spacers



```
struct Spacer_RelativeSpacing: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Spacer").font(.largeTitle)  
            Text("Relative Spacing").foregroundColor(.gray)  
            Text("You can add more spacers to create relative spacing in comparison to other  
                spacers.")  
            .frame(maxWidth: .infinity).padding()  
            .background(Color.yellow).foregroundColor(.black)  
        HStack(spacing: 50) {  
            VStack(spacing: 5) {  
                Spacer()  
                .frame(width: 5)  
                .background(Color.blue)  
                Text("33% Down")  
                Spacer()  
                .frame(width: 5)  
                .background(Color.blue)  
                Spacer()  
                .frame(width: 5)  
                .background(Color.blue)  
            }  
            VStack(spacing: 5) {  
                Spacer()  
                .frame(width: 5)  
                .background(Color.blue)  
                Spacer()  
            }  
        }  
    }  
}
```

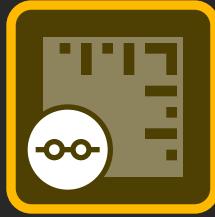
Spacers are views and can be modified like views.



```
.frame(width: 5)
.background(Color.blue)
Spacer()
    .frame(width: 5)
    .background(Color.blue)
Text("75% Down")
Spacer()
    .frame(width: 5)
    .background(Color.blue)
}
}
}.font(.title)
}
```

Note: You can also use Spacers horizontally to place views a percentage from the leading or trailing sides of the screen.

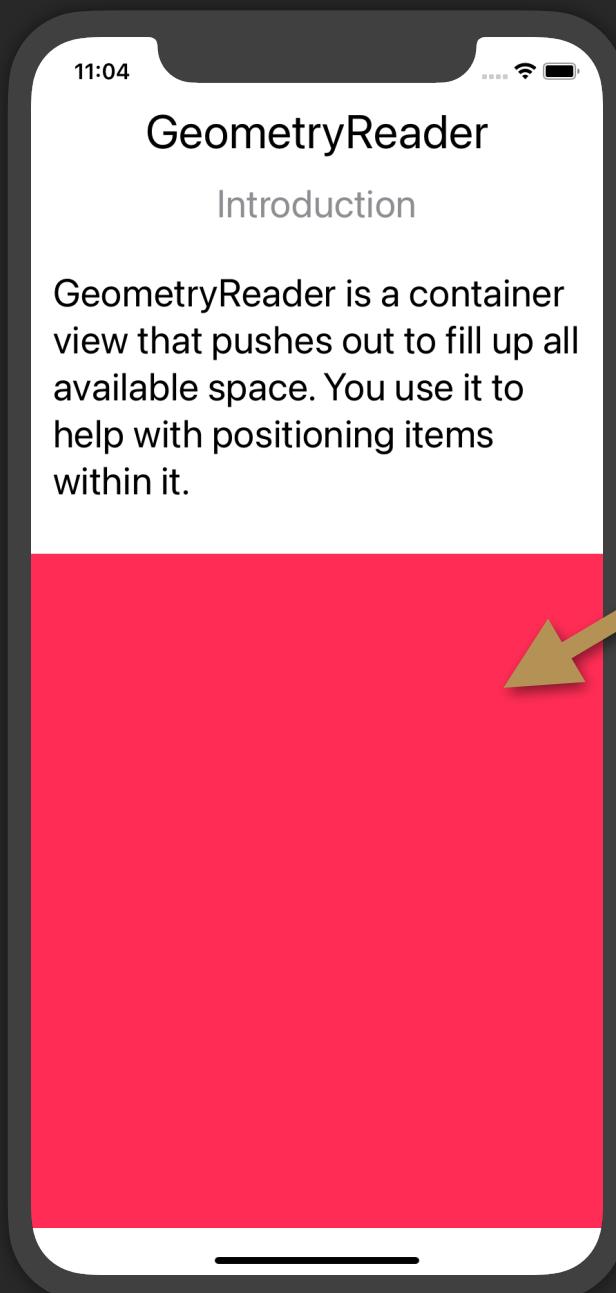
GeometryReader



It is difficult, if not impossible, to get the size of a view. This is where the GeometryReader can help.

The GeometryReader is similar to a push-out container view in that you can add child views to it. It will allow you to inspect and use properties that can help with positioning other views within it. You can access properties like height, width and safe area insets which can help you dynamically set the sizes of views within it so they look good on any size device.

Introduction



```
struct GeometryReader_Intro : View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("GeometryReader", subtitle: "Introduction", desc: "GeometryReader is a  
            container view that pushes out to fill up all available space. You use it to help with  
            positioning items within it.",  
            back: .clear)  
  
            GeometryReader { _ in  
                // No child views inside  
            }  
            .background(Color.pink)  
            .font(.title)  
        }  
    }  
}
```

In SwiftUI, when you see the word
"geometry", think size and position.

Alignment

11:06



GeometryReader

Alignment

Child views within the GeometryReader are aligned in the upper left corner by default.

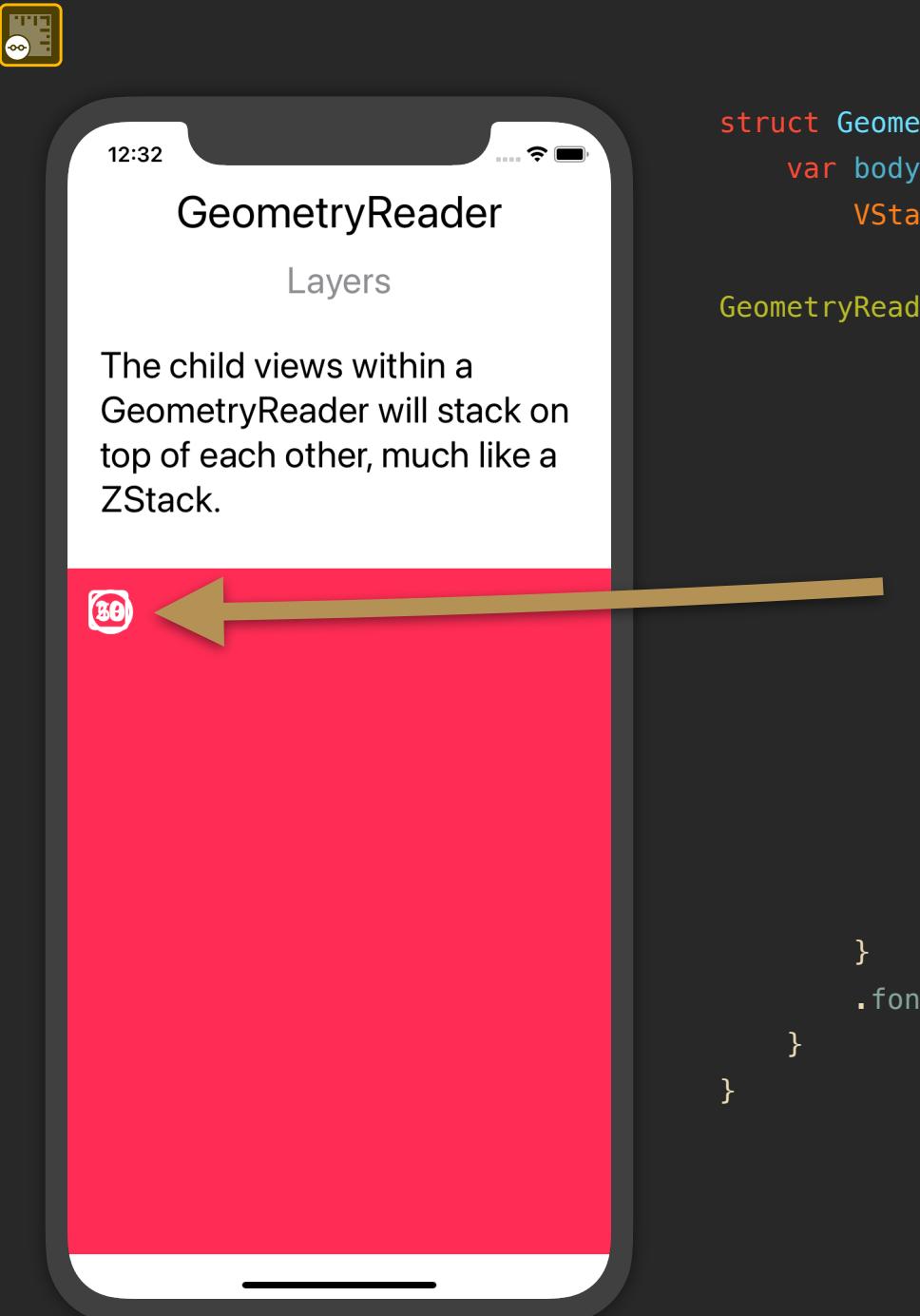


```
struct GeometryReader_Alignment: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("GeometryReader", subtitle: "Alignment", desc: "Child views within the  
GeometryReader are aligned in the upper left corner by default.", back: .clear)  
        }  
    }  
}
```

```
GeometryReader { _ in  
    Image(systemName: "arrow.up.left")  
        .padding()  
}  
.background(Color.pink)  
}  
.font(.title)  
}
```

Notice that there is no alignment or positioning specified on the image.

Layers

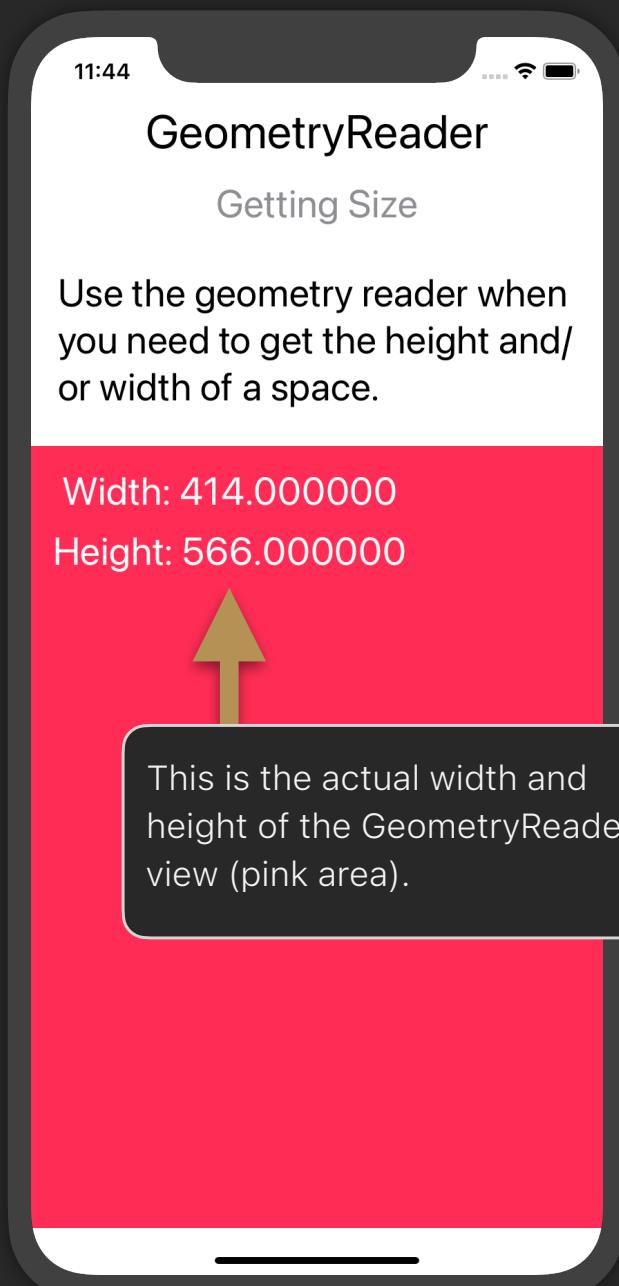


```
struct GeometryReader_Layers: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("GeometryReader", subtitle: "Layers", desc: "The child views within a  
GeometryReader will stack on top of each other, much like a ZStack.",  
back: .clear)  
  
            GeometryReader {_ in  
                Image(systemName: "18.circle")  
                    .padding()  
                Image(systemName: "20.square")  
                    .padding()  
                Image(systemName: "50.circle")  
                    .padding()  
            }  
            .font(.largeTitle)  
            .foregroundColor(.white)  
            .background(Color.pink)  
        }  
        .font(.title)  
    }  
}
```

Note, I wouldn't recommend using a GeometryReader in place of a ZStack.

ZStack provides convenient alignment options for layout that GeometryReader does not.

Getting Size



```
struct GeometryReader_GettingSize : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("GeometryReader", subtitle: "Getting Size", desc: "Use the geometry reader when you need to get the height and/or width of a space.", back: .clear)
        }
    }
}
```

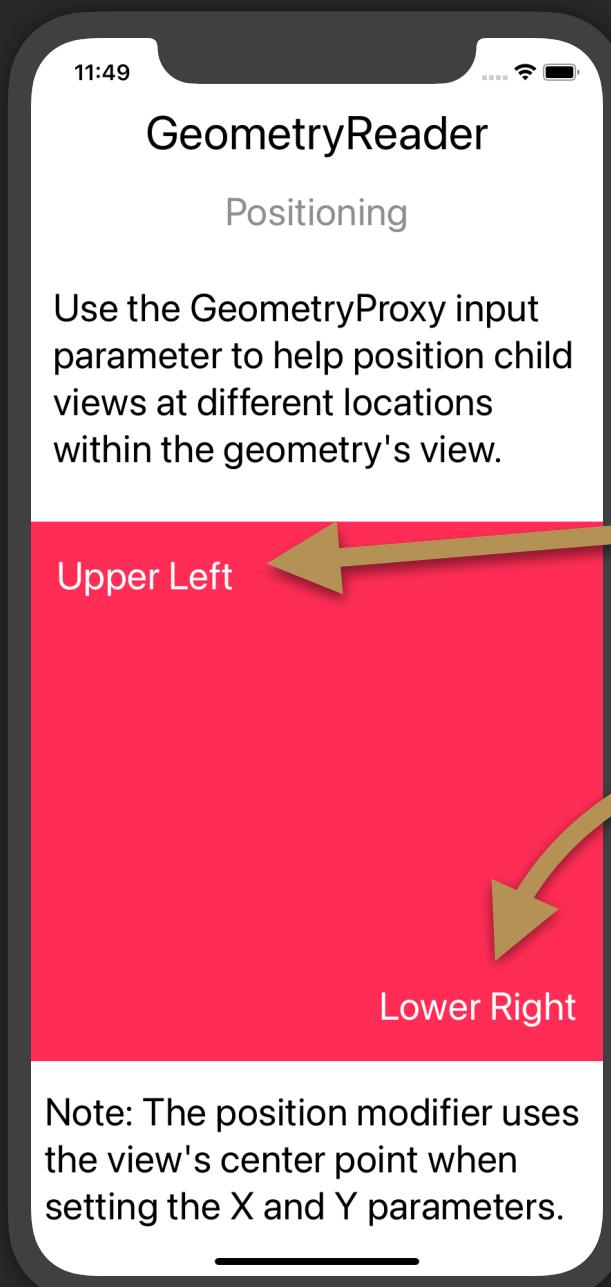
```
GeometryReader { geometryProxy in
    VStack(spacing: 10) {
        Text("Width: \(geometryProxy.size.width)")
        Text("Height: \(geometryProxy.size.height)")
    }
    .padding()
    .foregroundColor(.white)
}
.background(Color.pink)
.font(.title)
```

Define a parameter to reference the geometry's coordinate space from a "proxy".

The GeometryProxy is a representation of the GeometryReader's size and coordinate space.

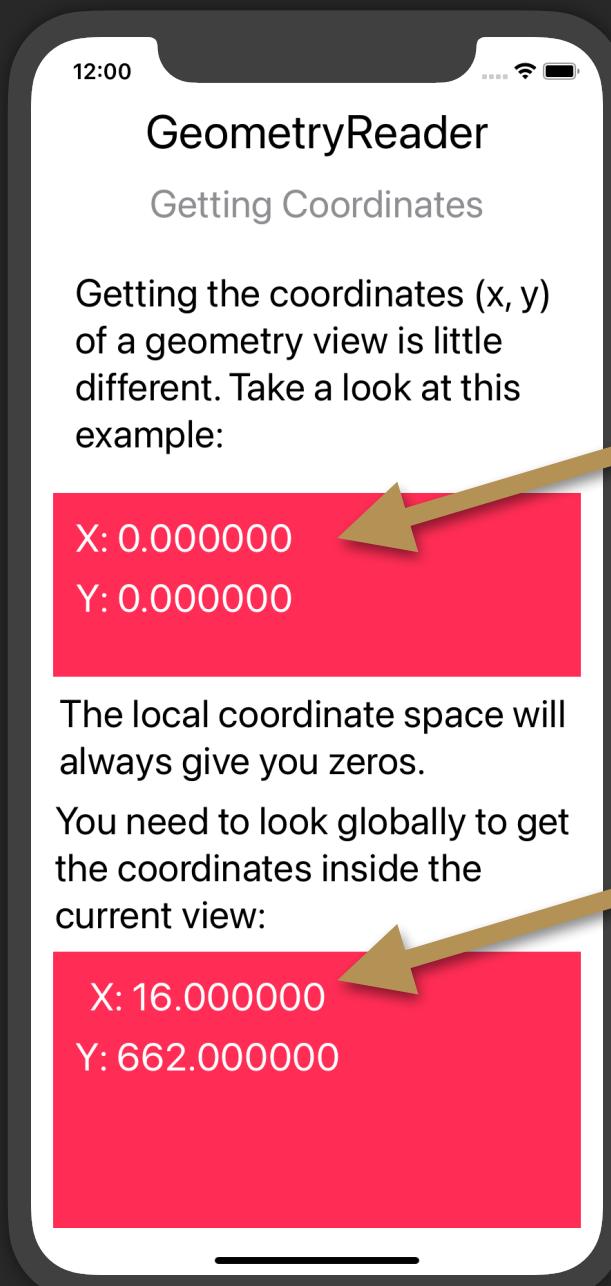
The `geometryProxy.size` will give you access to the height and width of the space the GeometryReader is taking up on the screen.

Positioning



```
struct GeometryReader_Positioning: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("GeometryReader").font(.largeTitle)  
            Text("Positioning").font(.title).foregroundColor(.gray)  
            Text("Use the GeometryProxy input parameter to help position child views at  
different locations within the geometry's view.")  
                .font(.title)  
                .padding()  
  
            GeometryReader { geometryProxy in  
                Text("Upper Left")  
                    .font(.title)  
                    .position(x: geometryProxy.size.width/5,  
                              y: geometryProxy.size.height/10)  
  
                Text("Lower Right")  
                    .font(.title)  
                    .position(x: geometryProxy.size.width - 90,  
                              y: geometryProxy.size.height - 40)  
            }  
            .background(Color.pink)  
            .foregroundColor(.white)  
  
            Text("Note: The position modifier uses the view's center point when setting the X  
and Y parameters.")  
                .font(.title)  
            }  
        }  
    }
```

Getting Coordinates



```
struct GeometryReader_GettingCoordinates : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("GeometryReader", subtitle: "Getting Coordinates", desc: "Getting the
coordinates (x, y) of a geometry view is little different. Take a look at this example:",
back: .clear)

            GeometryReader { geometryProxy in
                VStack(spacing: 10) {
                    Text("X: \(geometryProxy.frame(in: CoordinateSpace.local).origin.x)")
                    Text("Y: \(geometryProxy.frame(in: CoordinateSpace.local).origin.y)")
                }
                .foregroundColor(.white)
            }
            .background(Color.pink)

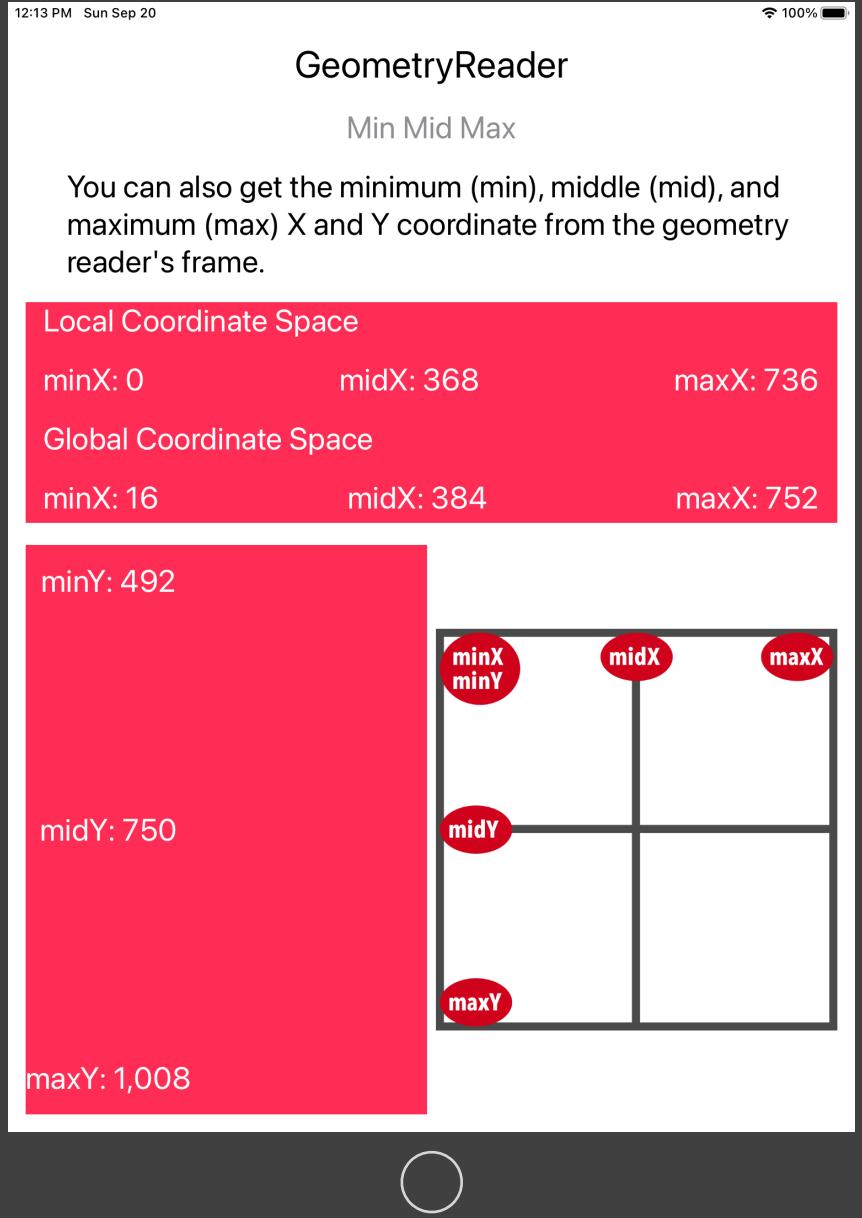
            Text("The local coordinate space will always give you zeros.")
            Text("You need to look globally to get the coordinates inside the current view:")
            GeometryReader { geometryProxy in
                VStack(spacing: 10) {
                    Text("X: \(geometryProxy.frame(in: .global).origin.x)")
                    Text("Y: \(geometryProxy.frame(in: .global).origin.y)")
                }
                .foregroundColor(.white)
            }
            .background(Color.pink)
            .frame(height: 200)
            .font(.title)
            .padding(.horizontal)
        }
    }
}
```

I left out "CoordinateSpace" in this example (it's optional).

The global coordinate space is the entire screen. We are looking at the origin of the geometry proxy's frame within the entire screen.



Min Mid Max Coordinates



```
struct GeometryReader_MinMidMax: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("GeometryReader", subtitle: "Min Mid Max", desc: "You can also get the minimum (min), middle (mid), and maximum (max) X and Y coordinate from the geometry reader's frame.", back: .clear)
            GeometryReader { geometry in
                VStack(alignment: .leading, spacing: 20) {
                    Text("Local Coordinate Space")
                    HStack(spacing: 10) {
                        // I'm converting to Int just so we don't have so many zeros
                        Text("minX: \(Int(geometry.frame(in: .local).minX))")
                        Spacer()
                        Text("midX: \(Int(geometry.frame(in: .local).midX))")
                        Spacer()
                        Text("maxX: \(Int(geometry.frame(in: .local).maxX))")
                    }
                    Text("Global Coordinate Space")
                    HStack(spacing: 10) {
                        // I'm converting to Int just so we don't have so many zeros
                        Text("minX: \(Int(geometry.frame(in: .global).minX))")
                        Spacer()
                        Text("midX: \(Int(geometry.frame(in: .global).midX))")
                        Spacer()
                        Text("maxX: \(Int(geometry.frame(in: .global).maxX))")
                    }
                }
            }.padding(.horizontal)
        }
    }
}
```

Min Mid Max Coordinates Continued



```

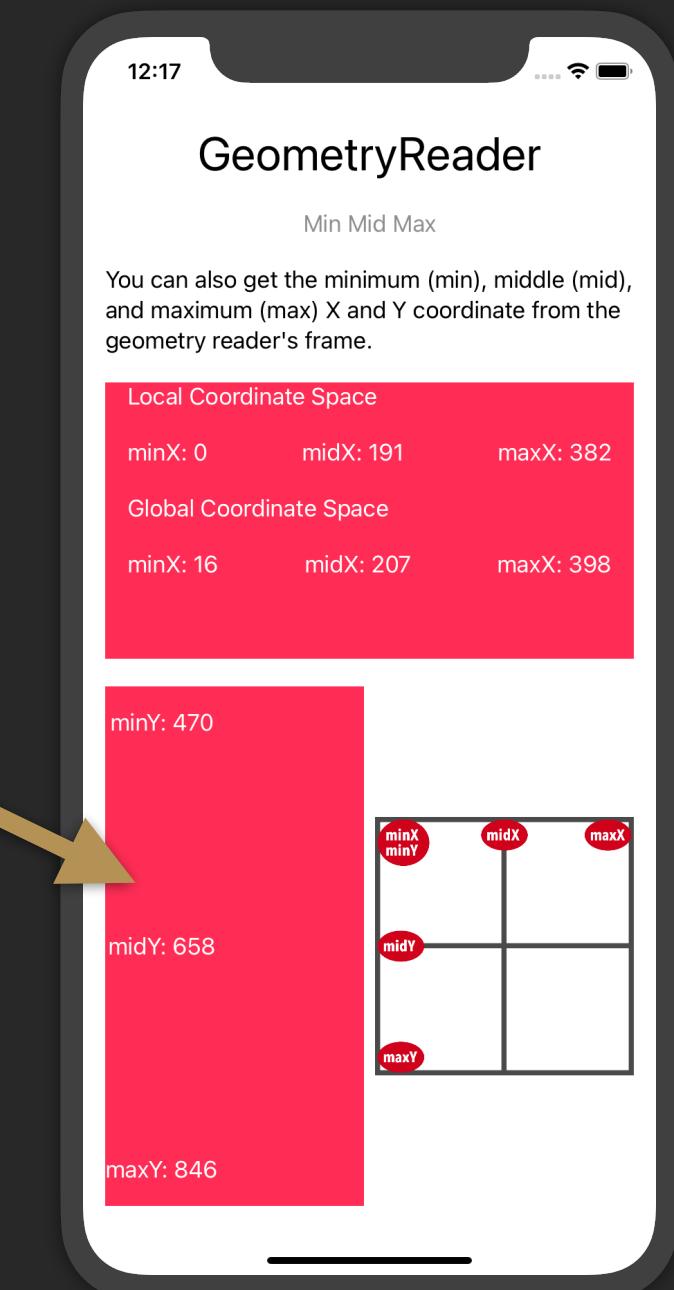
.frame(height: 200)
.foregroundColor(.white)
.background(Color.pink)

HStack {
    GeometryReader { geometry in
        VStack(spacing: 10) {
            Text("minY: \(Int(geometry.frame(in: .global).minY))")
            Spacer()
            Text("midY: \(Int(geometry.frame(in: .global).midY))")
            Spacer()
            Text("maxY: \(Int(geometry.frame(in: .global).maxY))")
        }.padding(.vertical)
    }
    .foregroundColor(.white)
    .background(Color.pink)

    Image("MinMidMax")
        .resizable()
        .aspectRatio(contentMode: .fit)
    }
    .font(.title)
    .padding()
}

```

Notice how the min, mid and max values change as the geometry reader adapts to different device sizes.



Safe Area Insets



GeometryReader

SafeAreaInsets

GeometryReader can also tell you the safe area insets it has.

```
geometryProxy.safeAreaInsets.leading: 48.000000  
geometryProxy.safeAreaInsets.trailing: 48.000000  
geometryProxy.safeAreaInsets.top: 0.000000  
geometryProxy.safeAreaInsets.bottom: 21.000000
```

```
HeaderView("GeometryReader", subtitle: "SafeAreaInsets", desc: "GeometryReader can also tell you the safe area insets it has.",  
back: .clear)  
  
GeometryReader { geometryProxy in  
    VStack {  
        Text("geometryProxy.safeAreaInsets.leading: \(geometryProxy.safeAreaInsets.leading)")  
        Text("geometryProxy.safeAreaInsets.trailing: \(geometryProxy.safeAreaInsets.trailing)")  
        Text("geometryProxy.safeAreaInsets.top: \(geometryProxy.safeAreaInsets.top)")  
        Text("geometryProxy.safeAreaInsets.bottom: \(geometryProxy.safeAreaInsets.bottom)")  
    }  
    .padding()  
}  
.background(Color.pink)  
.foregroundColor(.white)
```

LazyHGrid

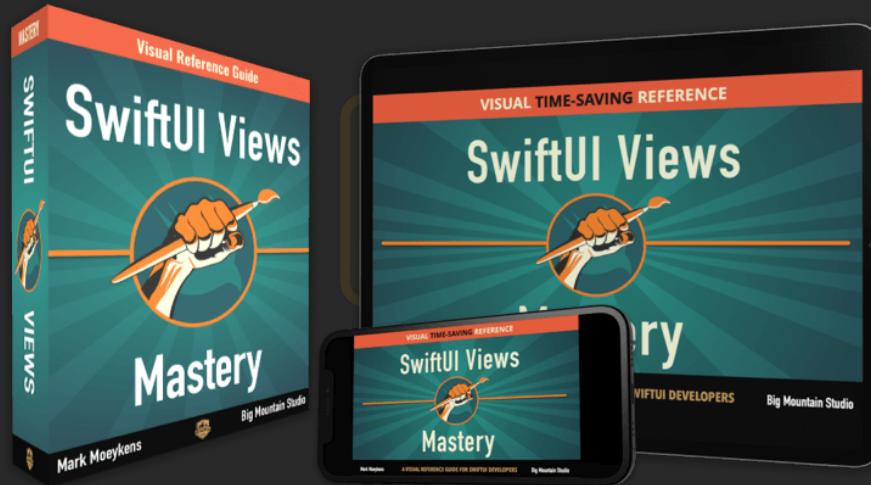


This SwiftUI chapter is locked in this
preview.

Similar to an HStack, the LazyHorizontal Grid will layout views horizontally but can be configured to use multiple rows and scroll off the screen. The word "lazy" here means that the child views are only created when SwiftUI needs them. This is called "lazy loading".

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

LazyVGrid



This SwiftUI chapter is locked in this
preview.

Similar to an HStack, the Lazy vertical Grid will layout views vertically but can be configured to use multiple columns and scroll off the screen. The word "lazy" here means that the child views are only created when SwiftUI needs them. This is called "lazy loading".

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ScrollViewReader



This SwiftUI chapter is locked in this
preview.

The Scroll View Reader gives you access to a function called `scrollTo`. With this function you can make a view within a scroll view visible by automatically scrolling it into view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ControlGroup



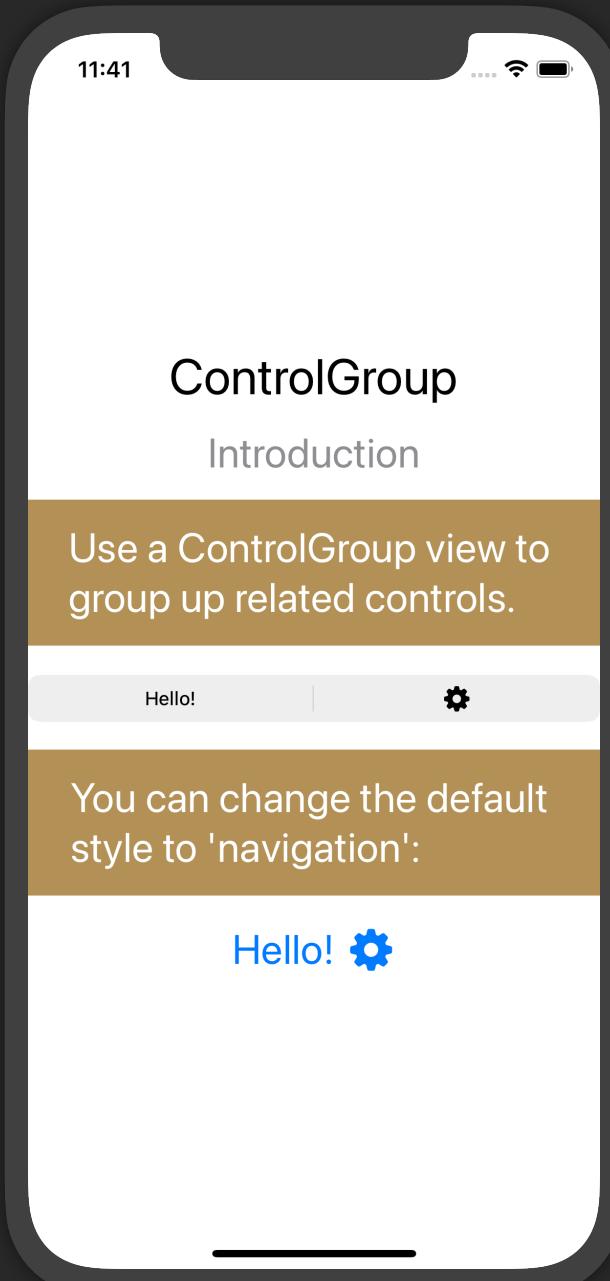
Use the ControlGroup to put similar types of controls together, such as buttons. In my opinion, the use of this seems limited.

This is a pull-in view.



Introduction

iOS 15



```
struct ControlGroup_Intro: View {
    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("ControlGroup",
                       subtitle: "Introduction",
                       desc: "Use a ControlGroup view to group up related controls.")

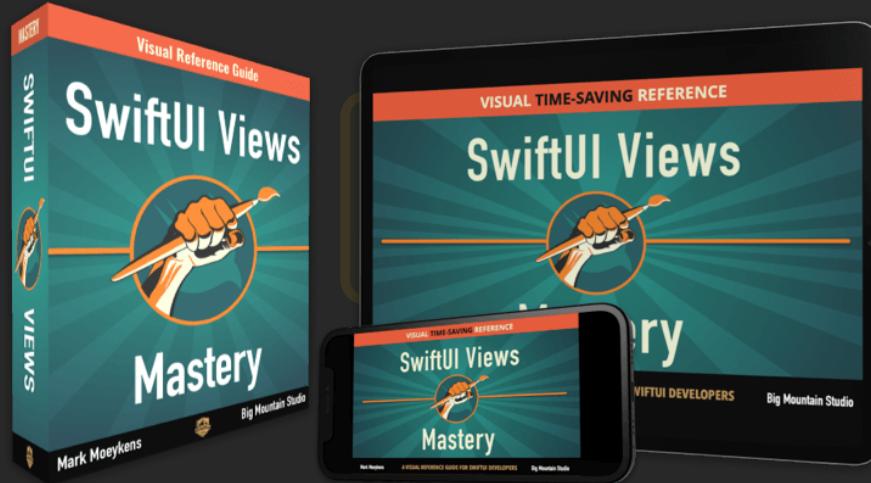
            ControlGroup {
                Button("Hello!") { }
                Button(action: {}) {
                    Image(systemName: "gearshape.fill")
                }
            }

            DescView(desc: "You can change the default style to 'navigation':")
            ControlGroup {
                Button("Hello!") { }
                Button(action: {}) {
                    Image(systemName: "gearshape.fill")
                }
            }
            .controlGroupStyle(.navigation)
        }
        .font(.title)
    }
}
```

Note: You may ask yourself when you would use this.

I think it makes more sense inside of toolbars as well as on macOS.

Table



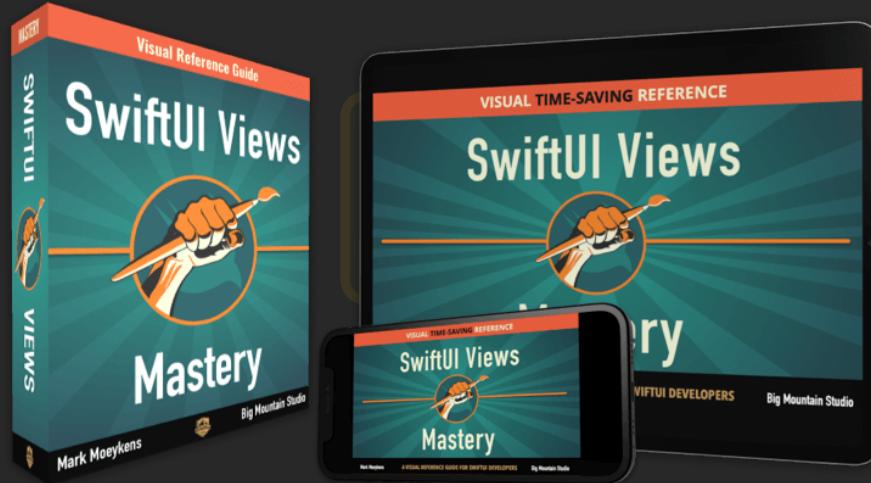
This SwiftUI chapter is locked in this
preview.

You can bind a collection of objects to a table and then each property to a table column.

In iOS though you will only see

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ViewThatFits



ViewThatFits is a very unique constraint that can contain multiple views but will only draw **one** of the views. That's right, it will only draw the view that fits the space it is given.



This SwiftUI chapter is locked in this preview.

You specify the larger view first, then progressively smaller views after that. The first view's size will be evaluated and if it can't fit in the given space, then the next view's size will be evaluated and so on.

If none of the views fit, the

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

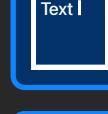
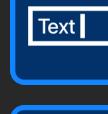
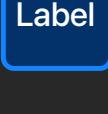
CONTROL VIEWS





Chapter Quick Links



 Button	 LabeledContent	 Picker	 SignInWithAppleButton
 Chart	 Link	 PhotosPicker	 Slider
 ColorPicker	 List	 ProgressView	 Stepper
 DatePicker	 Menu	 RenameButton	 TabView
 DisclosureGroup	 MultiDatePicker	 ScrollView	 TabView Paging
 Form	 NavigationStack	 Searchable	 Text
 Gauge	 NavigationSplitView	 SecureField	 TextEditor
 GroupBox	 NavLink	 Segmented Picker	 TextField
 Label	 OutlineGroup	 ShareLink	 Toggle

Button



The Button is a pull-in view with a wide range of composition and customization options to be presented to the user. The button can be just text, just an image or both combined.

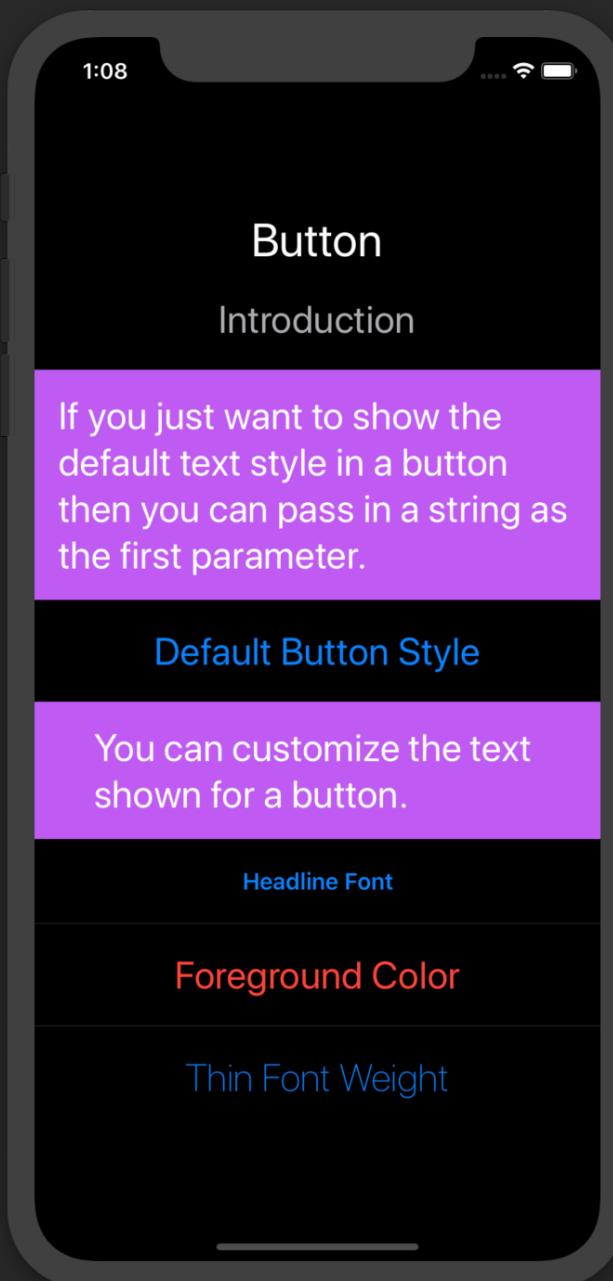
Introduction

```
VStack(spacing: 20) {  
    Text("Button")  
        .font(.largeTitle)  
    Text("Introduction")  
        .font(.title).foregroundColor(.gray)  
    Text("If you just want to show the default text style in a button then you can pass in a string as the first parameter")  
    ...  
    Button("Default Button Style") {  
        // Your code here  
    }  
    Text("You can customize the text shown for a button")  
    ...  
    Button(action: {  
        // Your code here  
    }) {  
        Text("Headline Font")  
            .font(.headline)  
    }  
    Divider()  
    Button(action: {}) {  
        Text("Foreground Color")  
            .foregroundColor(Color.red)  
    }  
    Divider()  
    Button(action: {}) {  
        Text("Thin Font Weight")  
            .fontWeight(.thin)  
    }  
}
```

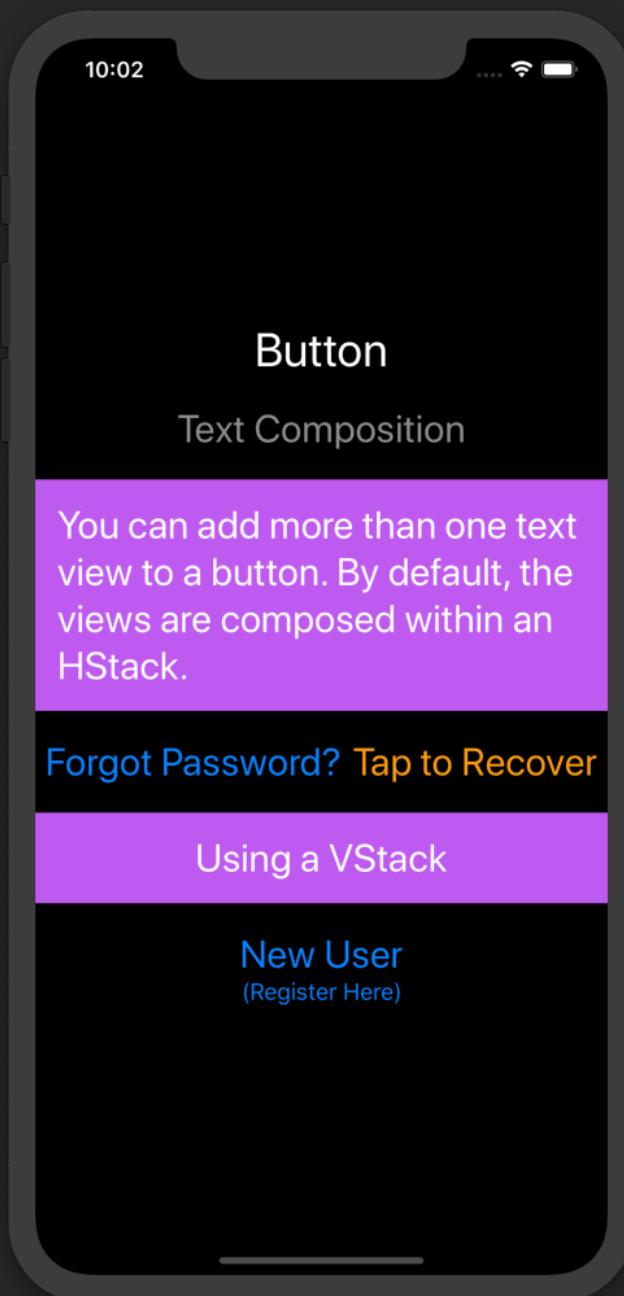
Default text-only buttons.

Use this initializer to customize the text.

For more text customization options, see the chapter on Text.



Text Composition



```
struct Button_TextModifiers : View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Button").font(.largeTitle)  
            Text("Text Composition").foregroundColor(.gray)  
            Text("You can add more than one text view to a button. By default, the views are  
                composed within an HStack.")  
            .padding().frame(maxWidth: .infinity)  
            .background(Color.purple)  
            .foregroundColor(.white).font(.title)  
  
            Button(action: {}, label: {  
                Text("Forgot Password?")  
                Text("Tap to Recover")  
                    .foregroundColor(.orange)  
            })  
  
            Text("Using a VStack")  
            .padding().frame(maxWidth: .infinity)  
            .background(Color.purple)  
            .foregroundColor(.white)  
  
            Button(action: {}, label: {  
                VStack {  
                    Text("New User")  
                    Text("(Register Here)").font(.body)  
                }  
            })  
            .font(.title)  
        }  
    }  
}
```

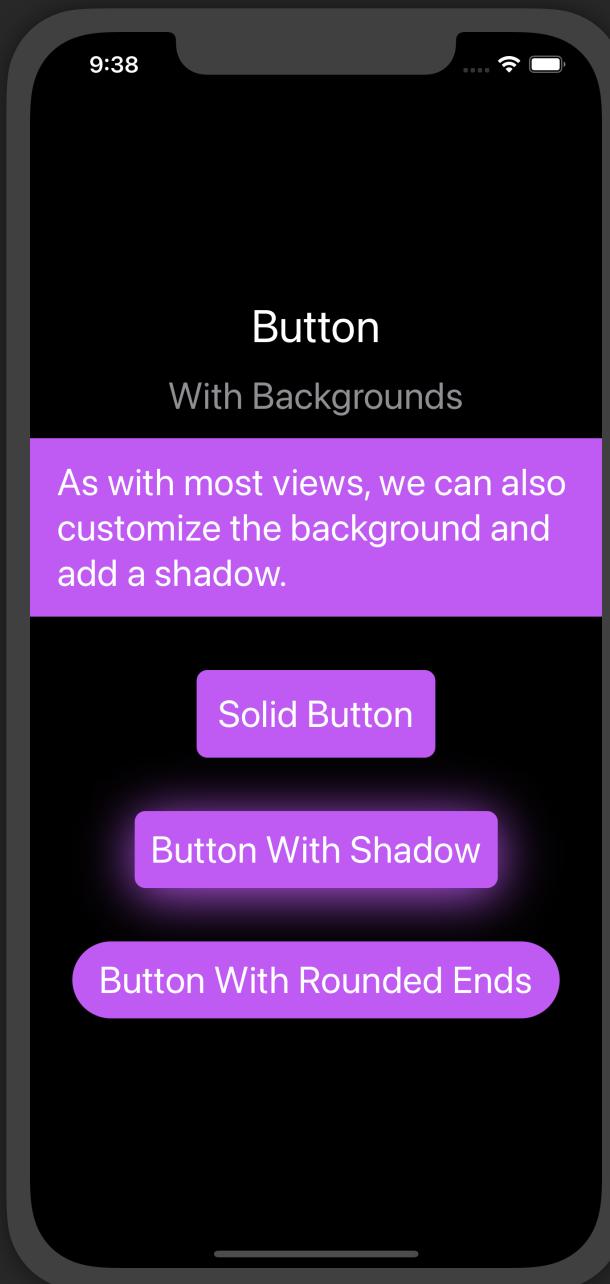
Views arranged horizontally by default.

With Backgrounds

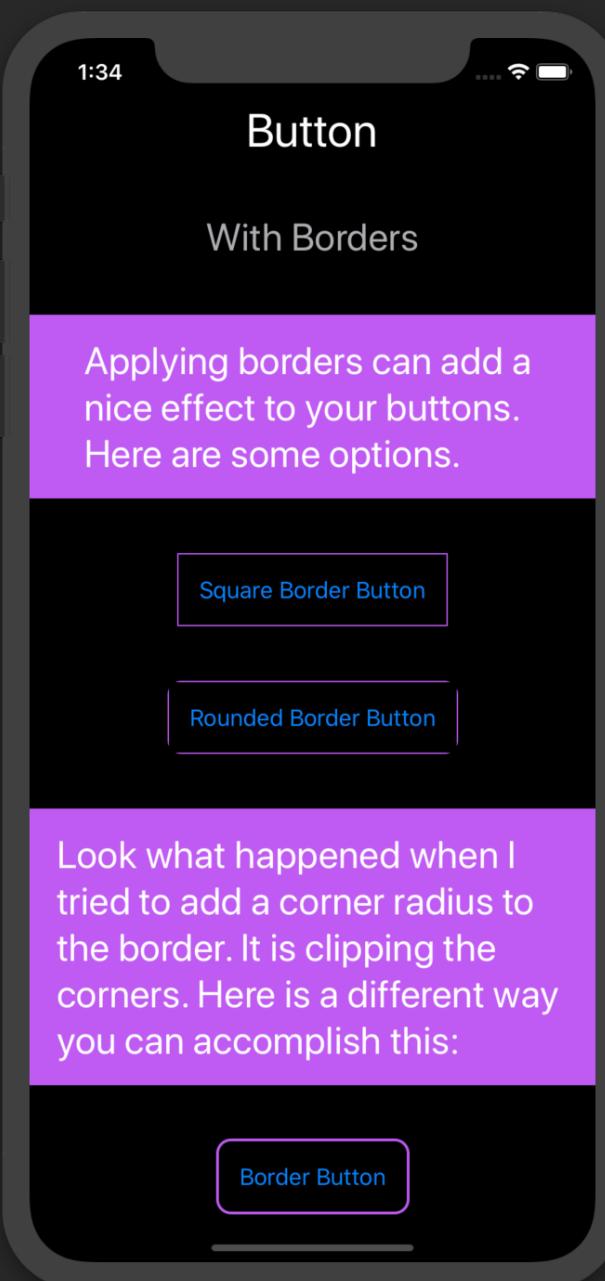
```
struct Button_EffectModifiers : View {  
    var body: some View {  
        VStack(spacing: 40) {  
            HeaderView("Button", subtitle: "With Backgrounds",  
                      desc: "As with most views, we can also customize the background and add a  
                      shadow.", back: .purple, textColor: .white)  
  
            Button(action: {}) {  
                Text("Solid Button")  
                    .padding()  
                    .foregroundColor(Color.white)  
                    .background(Color.purple)  
                    .cornerRadius(8)  
            }  
            Button(action: {}) {  
                Text("Button With Shadow")  
                    .padding(12)  
                    .foregroundColor(Color.white)  
                    .background(Color.purple)  
                    .cornerRadius(8)  
            }  
            .shadow(color: Color.purple, radius: 20, y: 5)  
  
            Button(action: {}) {  
                Text("Button With Rounded Ends")  
                    .padding(EdgeInsets(top: 12, leading: 20, bottom: 12, trailing: 20))  
                    .foregroundColor(Color.white)  
                    .background(Color.purple)  
                    .cornerRadius(100)  
            }  
        }  
        .font(.title)  
    }  
}
```

You will learn another way to do this using Shapes in the "Other Views" chapter.

A number higher than the height of the button will always give you a rounded ends.



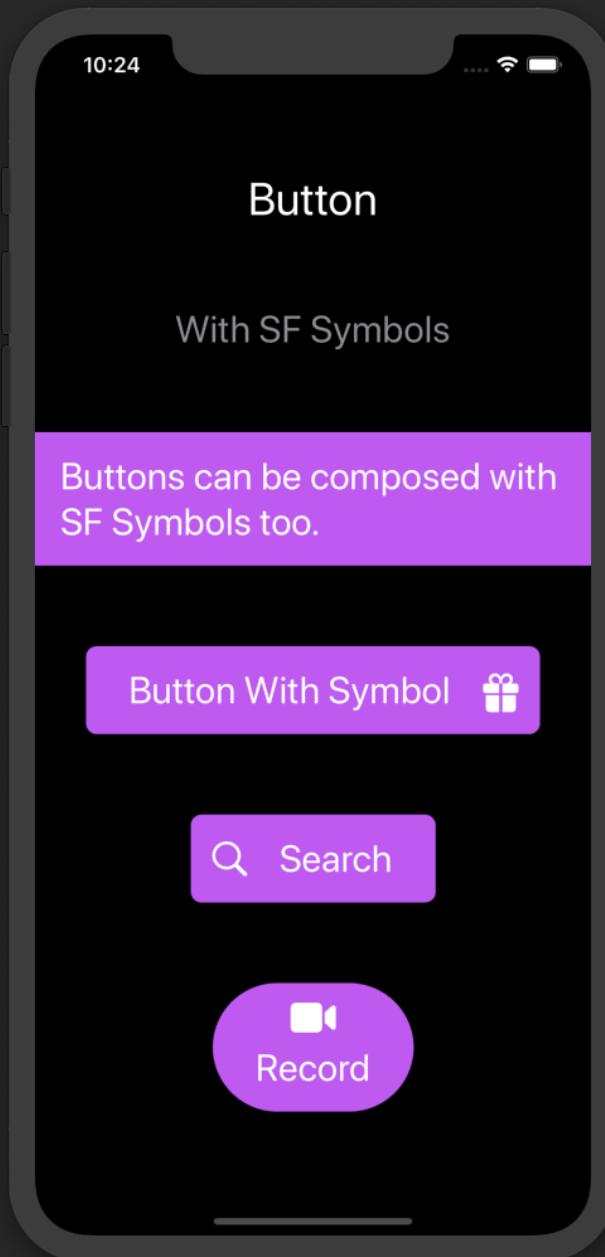
With Borders



```
Text("Button").font(.largeTitle)
Text("With Borders").font(.title).foregroundColor(.gray)
Text("Applying borders can add a nice effect to your buttons. Here are some options.")
    .padding().frame(maxWidth: .infinity)
    .background(Color.purple)
    .foregroundColor(.white).font(.title)

Button(action: {}) {
    Text("Square Border Button")
        .padding()
        .border(Color.purple)
}
Button(action: {}) {
    Text("Rounded Border Button")
        .padding()
        .border(Color.purple)
        .cornerRadius(10)
}
Text("Look what happened when I tried to add a corner radius to the border. It is clipping the corners. Here is a different way you can accomplish this:")
    ...
Button(action: {}) {
    Text("Border Button")
        .padding()
        .background(
            RoundedRectangle(cornerRadius: 10)
                .stroke(Color.purple, lineWidth: 2)
        )
}
```

With SF Symbols



```
Button(action: {}) {
    Text("Button With Symbol")
        .padding(.horizontal)
    Image(systemName: "gift.fill")
}
.padding()
.foregroundColor(Color.white)
.background(Color.purple)
.cornerRadius(8)

Button(action: {}) {
    Image(systemName: "magnifyingglass")
    Text("Search")
        .padding(.horizontal)
}
.padding()
.foregroundColor(Color.white)
.background(Color.purple)
.cornerRadius(8)

Button(action: {}) {
    VStack {
        Image(systemName: "video.fill")
        Text("Record")
            .padding(.horizontal)
    }
}
.padding()
.foregroundColor(Color.white)
.background(Color.purple)
.cornerRadius(.infinity)
```

For even more ways to customize buttons, see the chapter on **Paints** where you can learn how to apply the 3 different gradients to them.

With Images

```
struct Button_WithPhotos: View {  
    var body: some View {  
        VStack(spacing: 10) {  
            Text("Button")  
                .font(.largeTitle)  
              
            Button(action: {}) {  
                Image("yosemite")  
            }  
        }  
    }  
}
```

```
Button(action: {}) {  
    Image("yosemite")  
        .renderingMode(.original)  
        .cornerRadius(40)  
}
```

```
Button(action: {}) {  
    Image("yosemite")  
        .renderingMode(.template)  
        .cornerRadius(40)  
}  
}  
.font(.title)  
}
```

Button

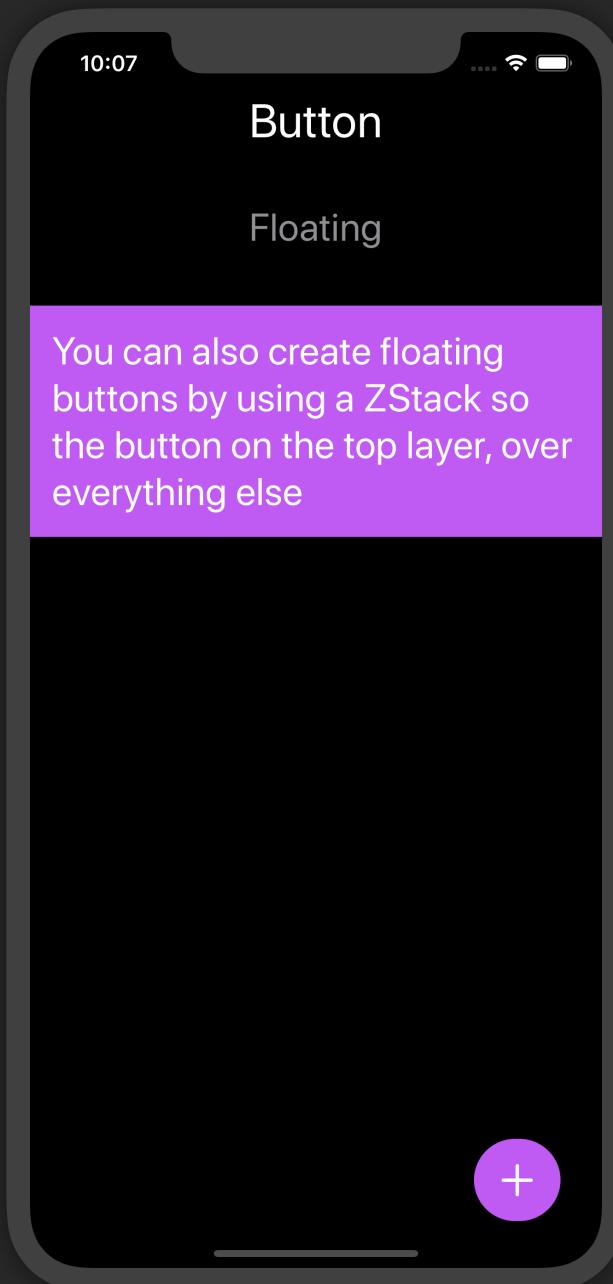


Earlier versions of iOS will render the image like you see in the last example.

If you are using a photo for the button content, it should show up fine from iOS 15+.

Before iOS 15 you might have to add the renderingMode modifier to make the image show.

Floating Action Button



```
ZStack {  
    VStack(spacing: 40) {  
        Text("Button")  
            .font(.largeTitle)  
  
        Text("Floating")  
            .font(.title).foregroundColor(.gray)  
  
        Text("You can also create floating buttons by using a ZStack so the button is on the top  
            layer, over everything else")  
        ...  
        Spacer()  
    }  
  
    VStack {  
        Spacer()  
        HStack {  
            Spacer()  
            Button(action: {}) {  
                Image(systemName: "plus")  
                    .font(.title)  
            }  
            .padding(20)  
            .foregroundColor(Color.white)  
            .background(Color.purple)  
            .cornerRadius(.infinity)  
        }  
        .padding(.trailing, 30) // Add 30 points on the trailing side of the button  
    }  
}
```

See the section on the Overlay modifier in the **Layout Modifiers** chapter for more ways to accomplish the same thing.

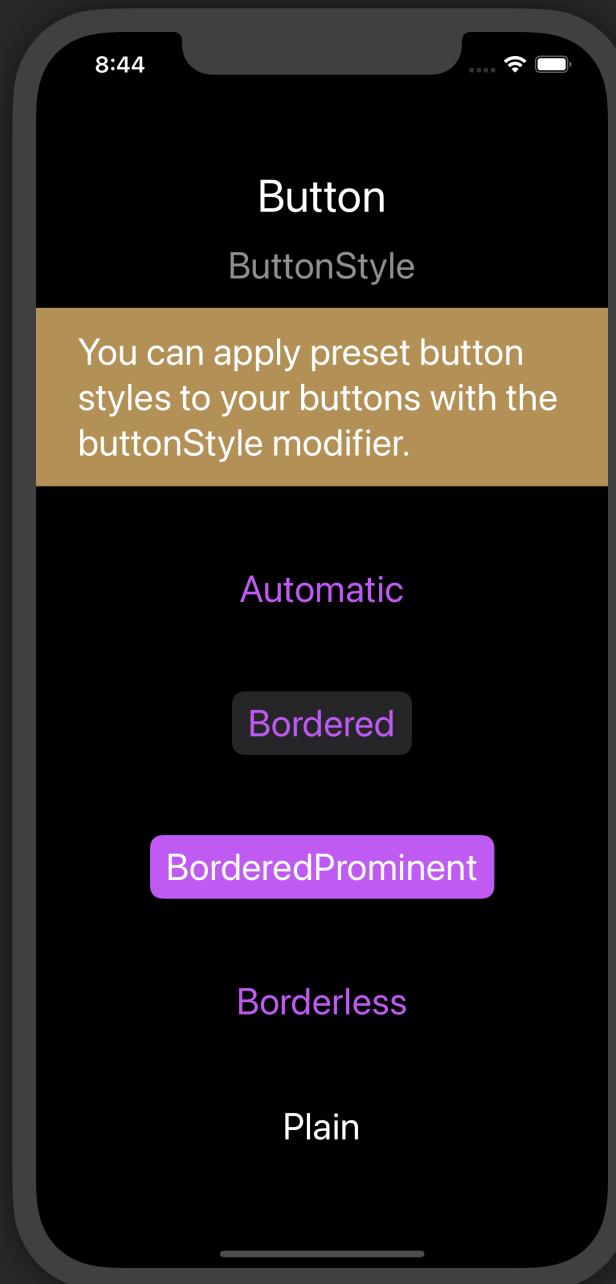


Button

iOS 15

ButtonStyle

Button

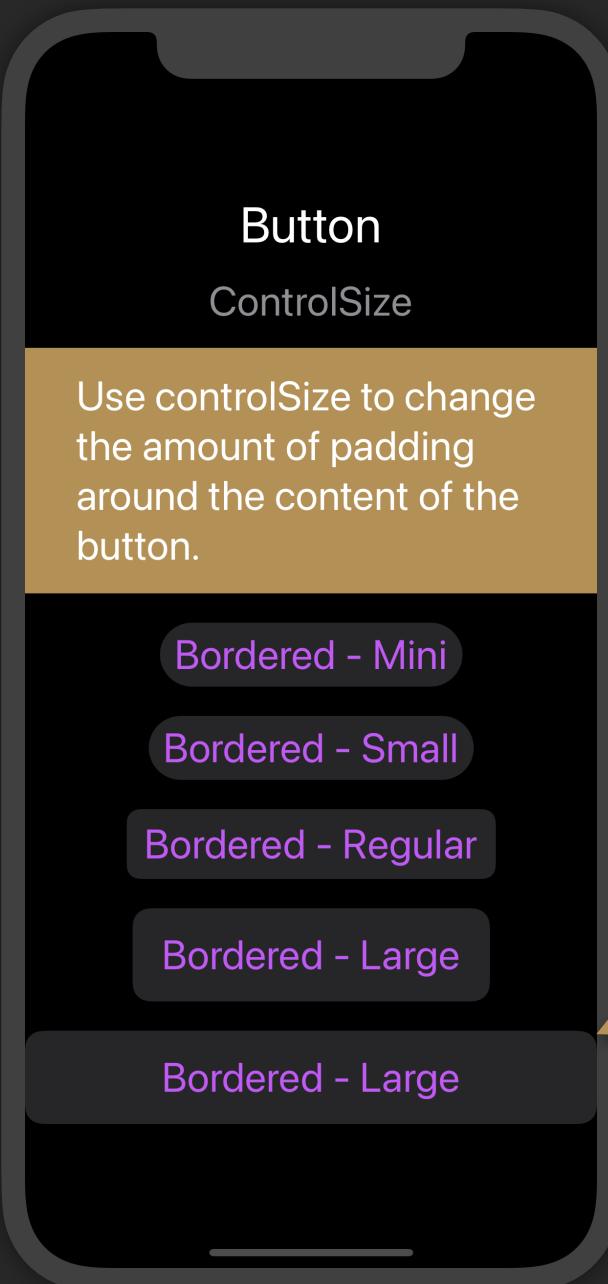


```
struct Button_ButtonStyle: View {  
    var body: some View {  
        VStack(spacing: 60.0) {  
            HeaderView("Button",  
                      subtitle: "ButtonStyle",  
                      desc: "You can apply preset button styles to your buttons with the  
                             buttonStyle modifier.")  
  
            Button("Automatic") { }  
                .buttonStyle(.automatic)  
  
            Button("Bordered") { }  
                .buttonStyle(.bordered)  
  
            Button("BorderedProminent") { }  
                .buttonStyle(.borderedProminent) ← Text becomes primary color.  
  
            Button("Borderless") { }  
                .buttonStyle(.borderless)  
  
            Button("Plain") { }  
                .buttonStyle(.plain) ← Accent color does not get applied.  
            }  
            .font(.title)  
            .accentColor(.purple)  
        }  
    }
```



ControlSize

iOS 15



Use controlSize to change the amount of padding around the content of the button.

```
struct Button_ControlSize: View {
    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("Button",
                       subtitle: "ControlSize",
                       desc: "Use controlSize to change the amount of padding around the content of the button.")

            Button("Bordered - Mini") { }
                .buttonStyle(.bordered)
                .controlSize(.mini)

            Button("Bordered - Small") { }
                .buttonStyle(.bordered)
                .controlSize(.small)

            Button("Bordered - Regular") { }
                .buttonStyle(.bordered)
                .controlSize(.regular)

            Button("Bordered - Large") { }
                .buttonStyle(.bordered)
                .controlSize(.large)

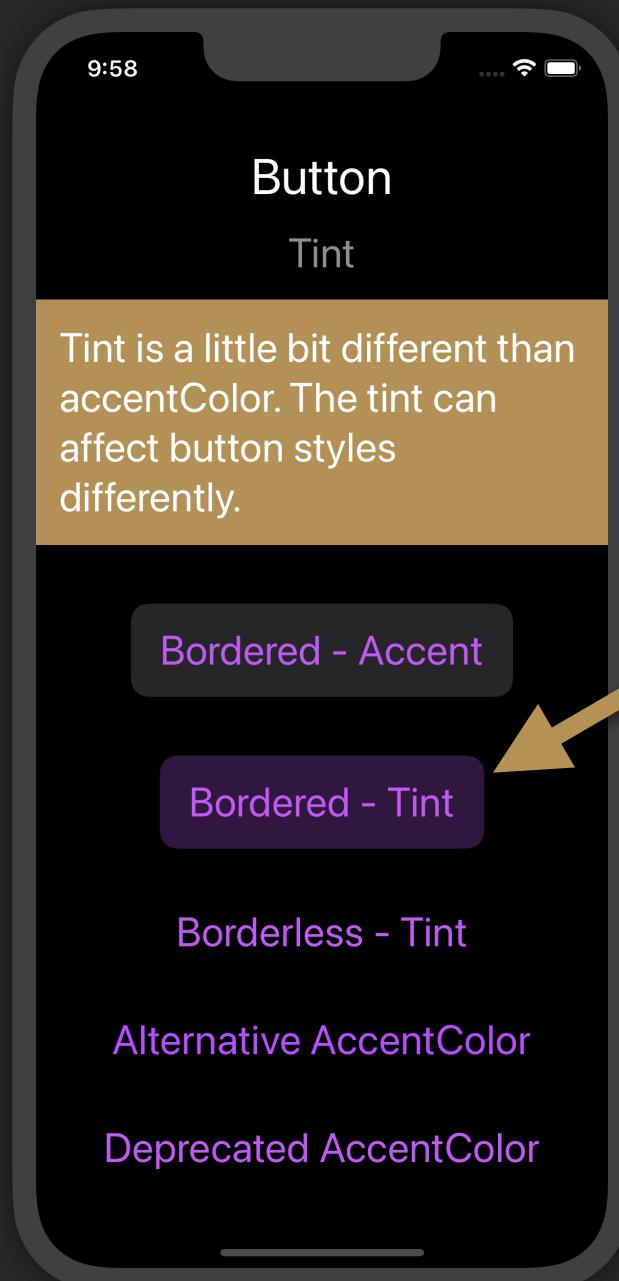
            Button(action: {}) {
                Text("Bordered - Large")
                    .frame(maxWidth: .infinity)
            }
                .buttonStyle(.bordered)
                .controlSize(.large)
        }
        .accentColor(.purple)
        .font(.title)
    }
}
```

You can still change the size manually and the shape will be the same.



Button

iOS 15



Tint

```
struct Button_Tint: View {  
    var body: some View {  
        VStack(spacing: 40) {  
            HeaderView("Button",  
                      subtitle: "Tint",  
                      desc: "Tint is a little bit different than accentColor. The tint can  
                            affect button styles differently.")  
  
            Button("Bordered – Accent") { }  
                .buttonStyle(.bordered)  
                .foregroundColor(Color.purple)  
  
            Button("Bordered – Tint") { }  
                .buttonStyle(.bordered)  
                .tint(.purple)  
  
            Button("Borderless – Tint") { }  
                .buttonStyle(.borderless)  
                .tint(.purple)  
  
            Button("Alternative AccentColor") { }  
                .buttonStyle(.borderless)  
                .foregroundColor(Color.purple)  
  
            Button("Deprecated AccentColor") { }  
                .buttonStyle(.borderless)  
                .accentColor(.purple)  
        }  
        .controlSize(.large)  
        .font(.title)  
    }  
}
```

Tint will affect the background of a bordered button and change the text color.

You can also use foregroundColor to change the color of button text.

Deprecation Note: The `accentColor` modifier will be deprecated.
Instead use:

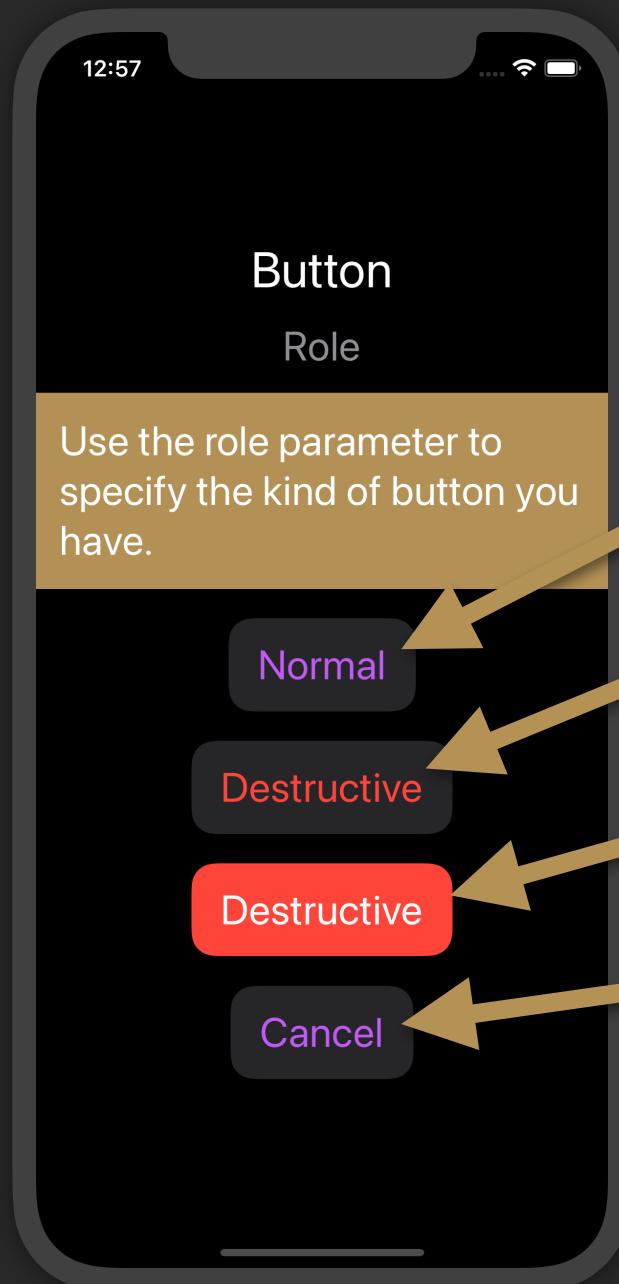
- `tint`
- `foregroundColor(Color.accentColor)`



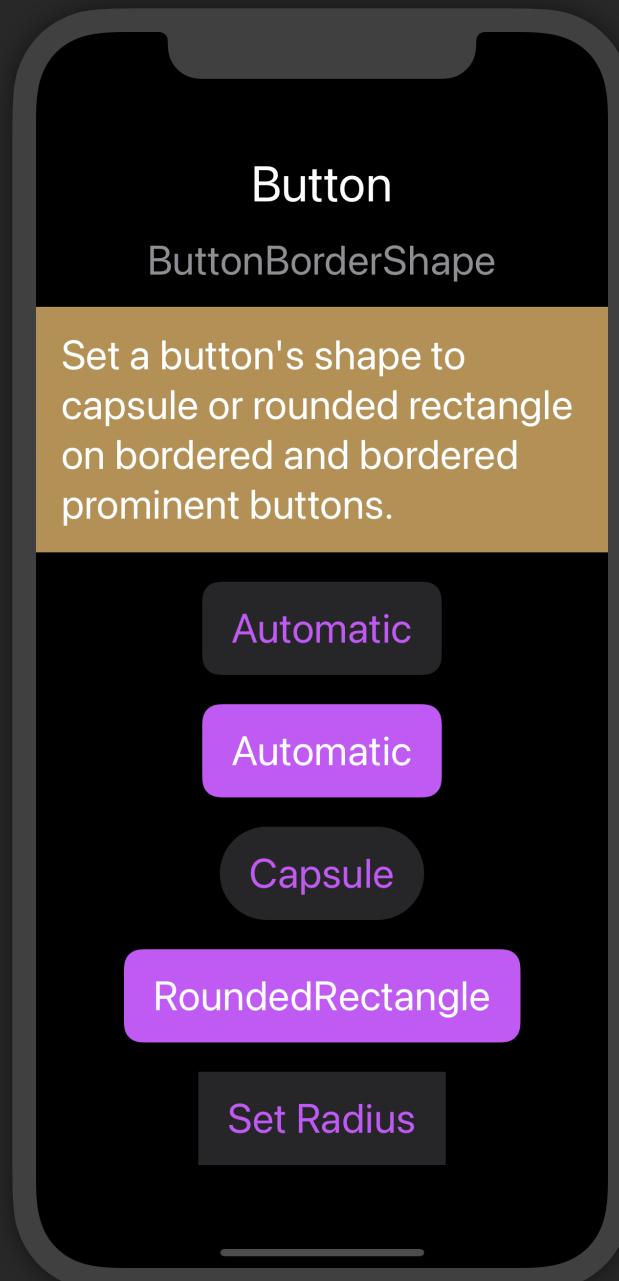
Button

Role

iOS 15



```
struct Button_Role: View {  
    var body: some View {  
        VStack(spacing: 20.0) {  
            HeaderView("Button",  
                      subtitle: "Role",  
                      desc: "Use the role parameter to specify the kind of button you have.")  
            Button("Normal") { }  
                .buttonStyle(.bordered)  
                .controlSize(.large)  
  
            Button("Destructive", role: .destructive) { }  
                .buttonStyle(.bordered)  
                .controlSize(.large)  
  
            Button("Destructive", role: .destructive) { }  
                .buttonStyle(.borderedProminent)  
                .controlSize(.large)  
  
            Button("Cancel", role: .cancel) { }  
                .buttonStyle(.bordered)  
                .controlSize(.large)  
        }  
        .font(.title)  
        .accentColor(.purple)  
    }  
}
```



ButtonBorderStyle

```
struct Button_ButtonBorderStyle: View {
    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("Button",
                       subtitle: "ButtonBorderStyle",
                       desc: "Set a button's shape to capsule or rounded rectangle on bordered and bordered prominent buttons.")

            Button("Automatic") { }
                .buttonStyle(.bordered)
                .buttonBorderShape(.automatic)
                .controlSize(.large)

            Button("Automatic") { }
                .buttonStyle(.borderedProminent)
                .buttonBorderShape(.automatic)
                .controlSize(.large)

            Button("Capsule") { }
                .buttonStyle(.bordered)
                .buttonBorderShape(.capsule)
                .controlSize(.large)

            Button("RoundedRectangle") { }
                .buttonStyle(.borderedProminent)
                .buttonBorderShape(.roundedRectangle)
                .controlSize(.large)

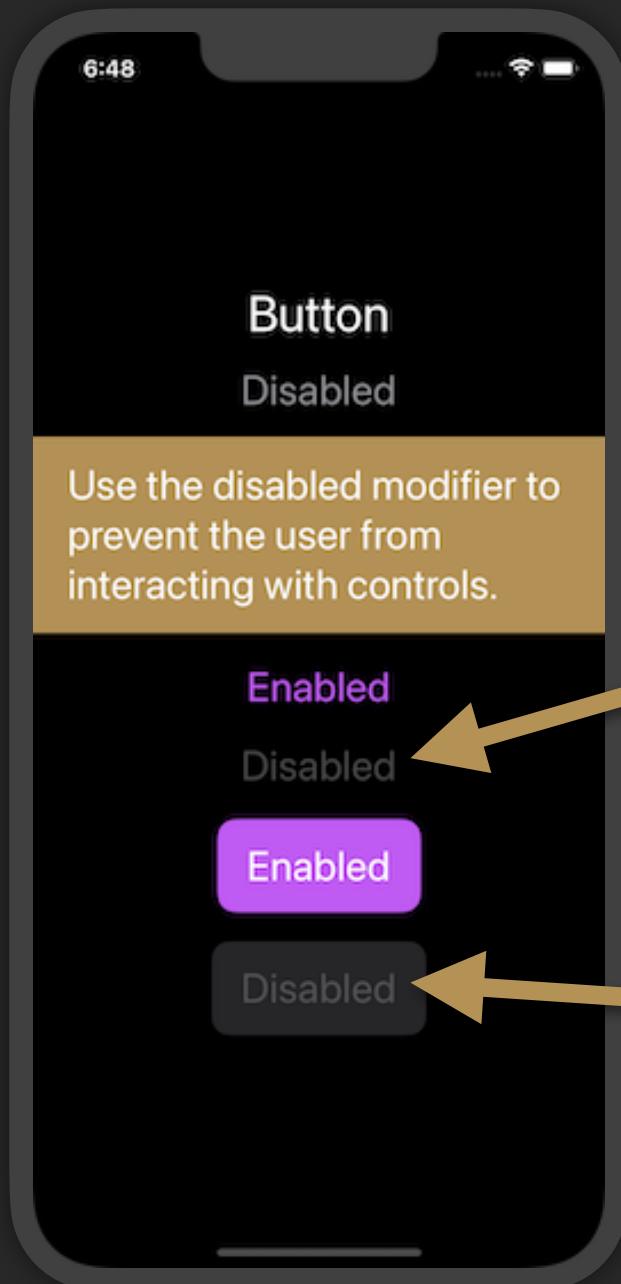
            Button("Set Radius") { }
                .buttonStyle(.bordered)
                .buttonBorderShape(.roundedRectangle(radius: 0))
                .controlSize(.large)
        }
        .font(.title)
        .accentColor(.purple)
    }
}
```

Note: This modifier **ONLY** works on buttons that are bordered or borderedProminent.

You can use the radius option to create rectangle backgrounds too by setting it to zero.

Disabled

```
struct Button_Disabled: View {  
    var body: some View {  
        VStack(spacing: 20.0) {  
            HeaderView("Button",  
                subtitle: "Disabled",  
                desc: "Use the disabled modifier to prevent the user from interacting  
                with buttons."  
  
            Button("Enabled") { }  
            .controlSize(.large)  
  
            Button("Disabled") { }  
            .controlSize(.large)  
            .disabled(true)  
  
            Button("Enabled") { }  
            .buttonStyle(.borderedProminent)  
            .controlSize(.large)  
  
            Button("Disabled") { }  
            .buttonStyle(.borderedProminent)  
            .controlSize(.large)  
            .disabled(true)  
        }  
        .font(.title)  
        .tint(.purple)  
    }  
}
```



Chart



This SwiftUI content is locked in this
preview.

A Chart is a container view. It contains “marks” (representations of values) which can be drawn in a variety of different ways. There are many different ways you can configure the chart and marks. This chapter will cover many of those options.

This is a push-out view.

[UNLOCK THE BOOK TODAY FOR ONLY \\$55!](#)

ColorPicker



This SwiftUI chapter is locked in this
preview.

The ColorPicker control allows you to give users the ability to select a color. This could be useful if you want to allow users to set the color of visual elements on the user interface.

This is a push-out horizontal

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

DatePicker



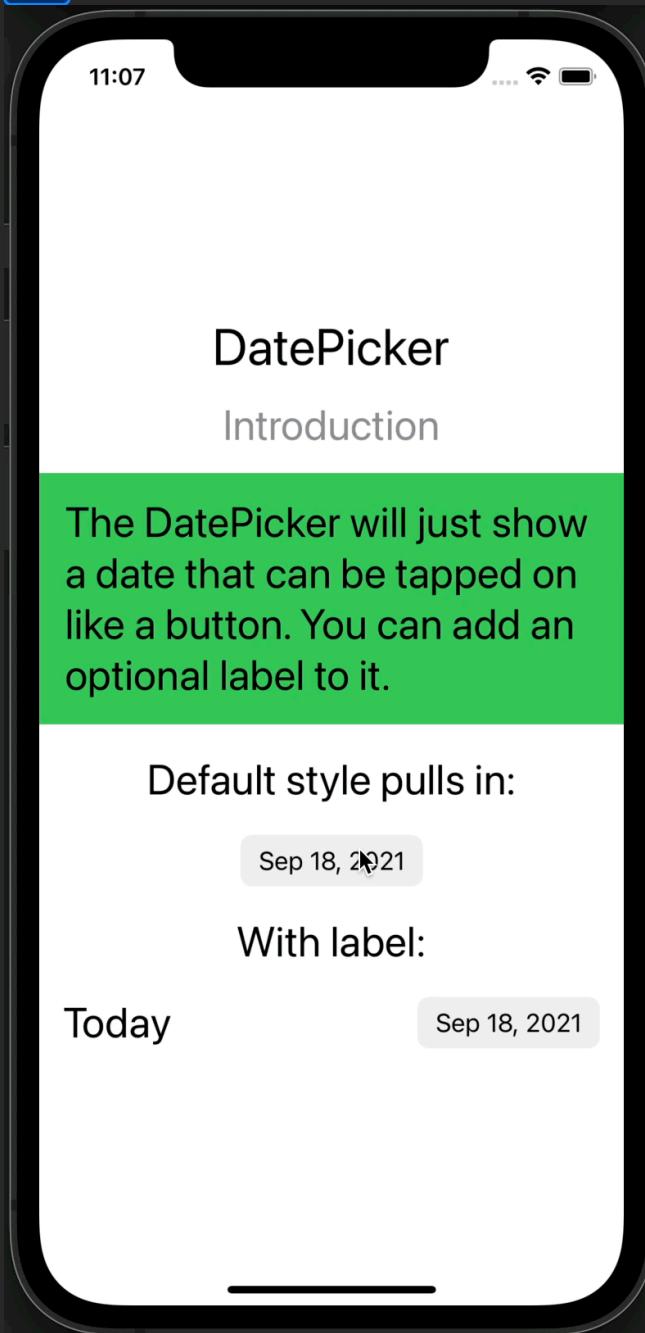
The date picker provides a way for the user to select a date and time. You bind the selected date to a property. You can read this property to find out what was selected or set this property for the DatePicker to show the date you want.
(Note: If you have to support the DatePicker for iOS 13, then it will look different from what you see in this chapter.)

This is a push-out view.



Introduction

```
struct DatePicker_Intro: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("DatePicker",  
                      subtitle: "Introduction",  
                      desc: "The DatePicker will just show a date that can be tapped on like a  
button. You can add an optional label to it.", back: .green)  
  
            Text("Default style pulls in:")  
  
            DatePicker("Today", selection: $date, displayedComponents: .date)  
                .labelsHidden()  
                .padding(.horizontal)  
  
            Text("With label:")  
  
            DatePicker("Today", selection: $date, displayedComponents: .date)  
                .padding(.horizontal)  
                .font(.title)  
        }  
    }  
}
```



Text("Default style pulls in:")

```
DatePicker("Today", selection: $date, displayedComponents: .date)  
    .labelsHidden()  
    .padding(.horizontal)
```

Hiding the label makes this view pull in.

Text("With label:")

```
DatePicker("Today", selection: $date, displayedComponents: .date)  
    .padding(.horizontal)
```

What you see here is representative of the **compact** date picker style
(text representation of the date).

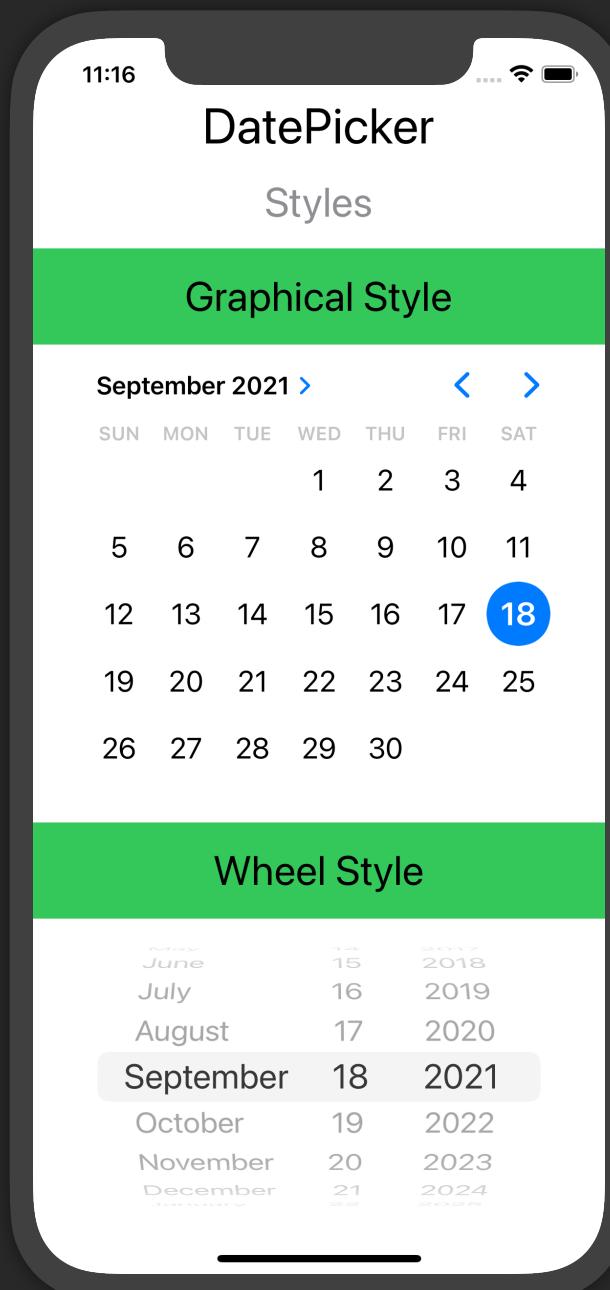
There are other styles available...



DatePicker

Styles

iOS 14



```
struct DatePicker_Styles: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 0) {  
            HeaderView("DatePicker",  
                      subtitle: "Styles",  
                      desc: "Graphical Style", back: .green)  
  
            DatePicker("Birthday", selection: $date, displayedComponents: .date)  
                .datePickerStyle(.graphical)  
                .frame(width: 320)  
  
            DescView(desc: "Wheel Style", back: .green)  
            DatePicker("Birthday", selection: $date, displayedComponents: .date)  
                .datePickerStyle(.wheel)  
                .labelsHidden()  
  
        }  
        .font(.title)  
        .ignoresSafeArea(edges: .bottom)  
    }  
}
```

Notice we didn't have to hide the labels on the graphical style. It's not shown. (But you should keep it set for accessibility purposes.)



For datePickerStyle, use:

< iOS 15

GraphicalDatePickerStyle()
WheelDatePickerStyle()

iOS 15+

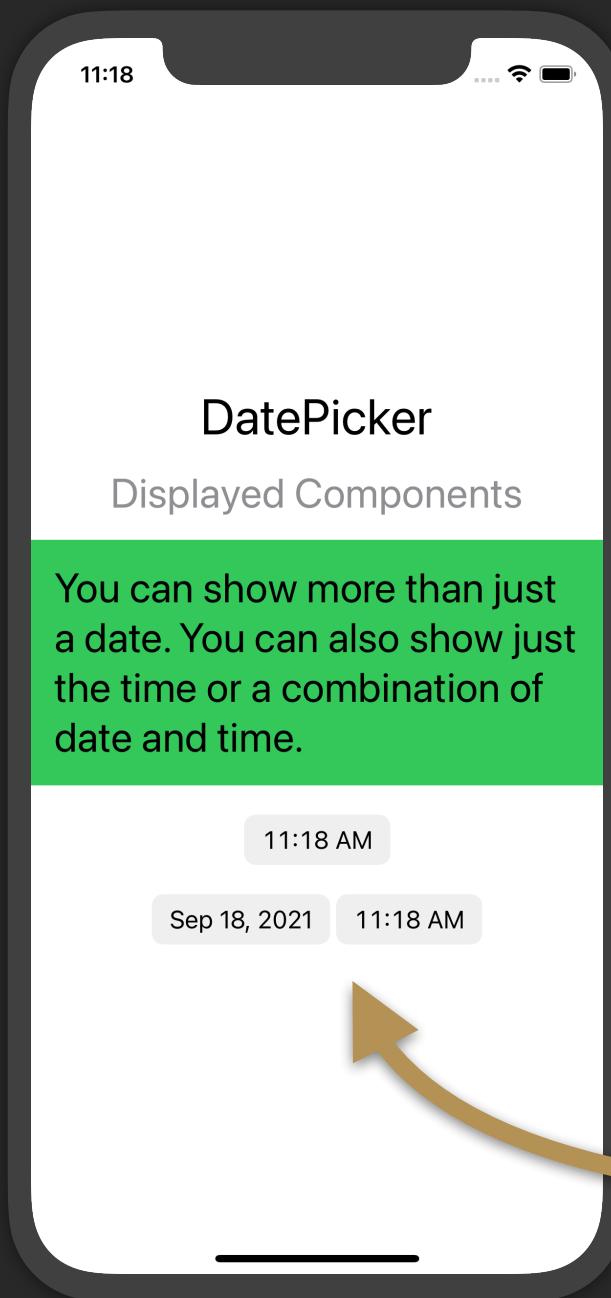
.graphical
.wheel



Displayed Components

```
struct DatePicker14_DisplayedComponents: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("DatePicker - iOS 14+",  
                      subtitle: "Displayed Components",  
                      desc: "You can show more than just a date. You can also show just  
                            the time or a combination of date and time.", back: .green)  
  
            DatePicker("Today", selection: $date, displayedComponents: .hourAndMinute)  
                .labelsHidden()  
                .padding(.horizontal)  
  
            DatePicker("Today", selection: $date, displayedComponents: [.hourAndMinute, .date])  
                .labelsHidden()  
                .padding(.horizontal)  
                .buttonStyle(.bordered)  
        }  
        .font(.title)  
    }  
}
```

Note that the order of the displayed components does not affect the displayed order. The hour and minute still come second.





Displayed in Form

11:20

DatePicker

Used in a Form

When used in a form, the date picker uses the compact styling by default.

Today

Sep 18, 2021

Graphical Picker Style:

September 2021 > < >						
SUN	MON	TUE	WED	THU	FRI	SAT
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

```
struct DatePicker_InForm: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("DatePicker",  
                      subtitle: "Used in a Form",  
                      desc: "When used in a form, the date  
picker uses the compact styling by default.",  
                      back: .green)  
  
            Form {  
                DatePicker("Today", selection: $date,  
                           displayedComponents: .date)  
  
                Section {  
                    Text("Graphical Picker Style:")  
                    DatePicker("Birthday", selection: $date,  
                               displayedComponents: .date)  
                    .datePickerStyle(.graphical)  
                }  
            }  
            .font(.title)  
        }  
    }  
}
```

When the compact style is tapped, a pop up shows the graphical date picker.

11:20

DatePicker

Used in a Form

When used in a form, the date picker uses the compact styling by default.

Today

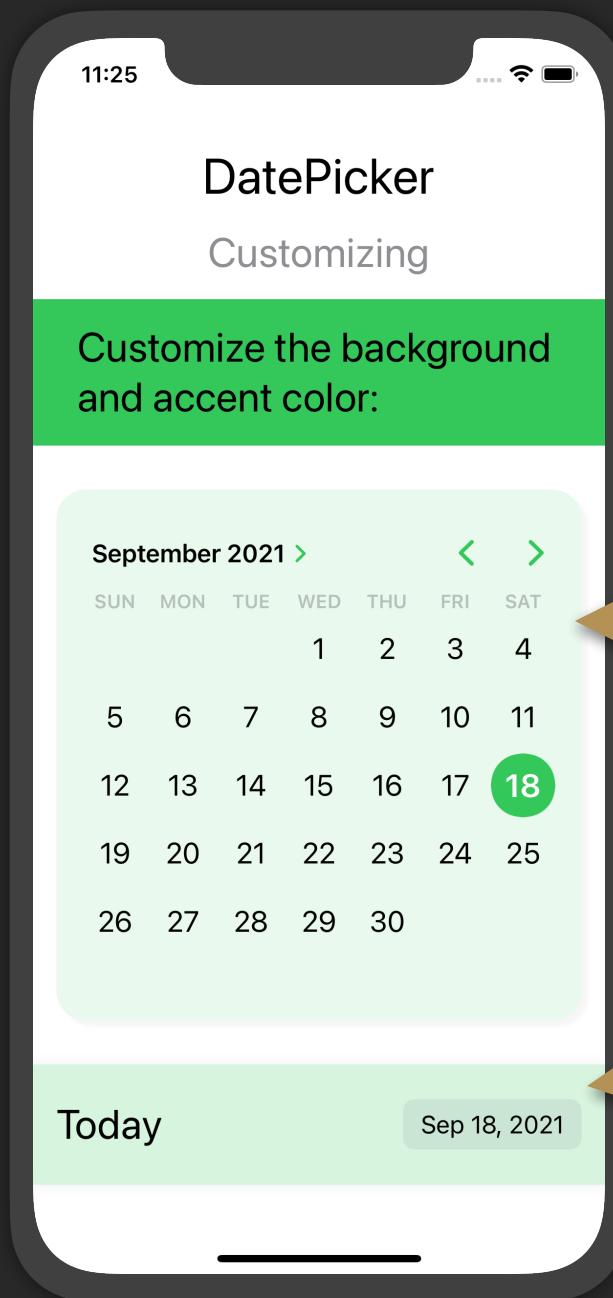
Sep 18, 2021

September 2021 > < >						
SUN	MON	TUE	WED	THU	FRI	SAT
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		



Customizing

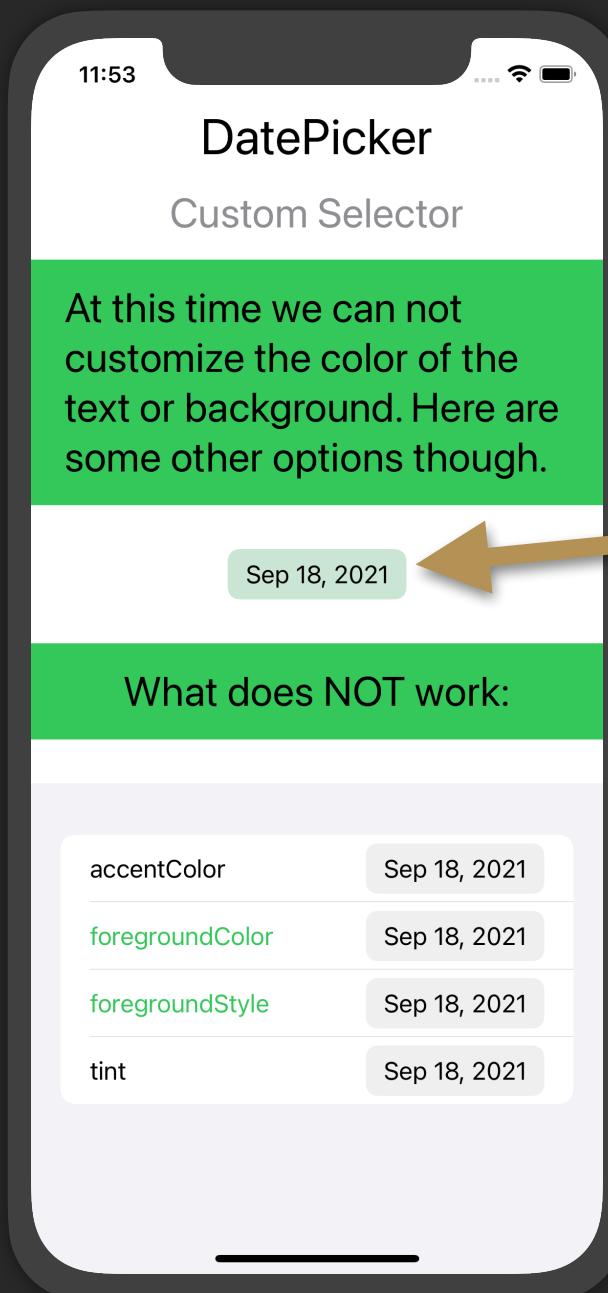
iOS 14



```
struct DatePicker_Customizing: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 30) {  
            HeaderView("DatePicker",  
                      subtitle: "Customizing",  
                      desc: "Customize the background and accent color:", back: .green)  
  
            DatePicker("Birthday", selection: $date, displayedComponents: .date)  
                .datePickerStyle(.graphical)  
                .accentColor(.green)  
                .padding()  
                .background(RoundedRectangle(cornerRadius: 20)  
                            .fill(Color.green)  
                            .opacity(0.1)  
                            .shadow(radius: 1, x: 4, y: 4))  
                .padding(.horizontal)  
  
            DatePicker("Today", selection: $date, displayedComponents: .date)  
                .frame(height: 50)  
                .padding()  
                .background(Rectangle()  
                            .fill(Color.green)  
                            .shadow(radius: 4)  
                            .opacity(0.2))  
        }  
        .font(.title)  
    }  
}
```



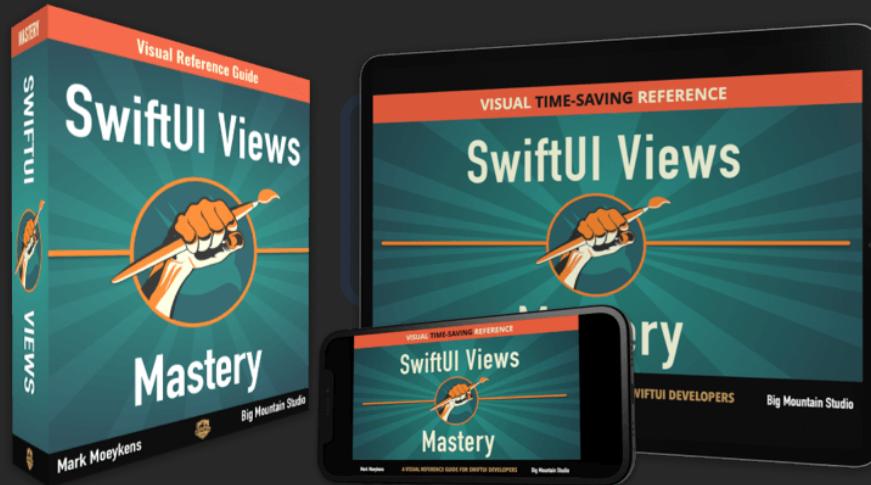
Custom Selector



```
struct DatePicker_CustomSelector: View {  
    @State private var date = Date()  
  
    var body: some View {  
        VStack(spacing: 30) {  
            HeaderView("DatePicker",  
                      subtitle: "Custom Selector",  
                      desc: "At this time we can not customize the color of the text or  
                             background. Here are some other options though.", back: .green)  
  
            DatePicker("Today", selection: $date, displayedComponents: .date)  
                .labelsHidden()  
                .background(RoundedRectangle(cornerRadius: 8, style: .continuous)  
                            .fill(Color.green).opacity(0.2))  
  
            DescView(desc: "What does NOT work:", back: .green)  
            Form {  
                DatePicker("accentColor", selection: $date, displayedComponents: .date)  
                    .accentColor(.green)  
  
                DatePicker("foregroundColor", selection: $date, displayedComponents: .date)  
                    .foregroundColor(.green)  
  
                DatePicker("foregroundStyle", selection: $date, displayedComponents: .date)  
                    .foregroundStyle(.green, .green, .green)  
  
                DatePicker("tint", selection: $date, displayedComponents: .date)  
                    .tint(.green)  
            }  
            .font(.body)  
        }  
        .font(.title)  
    }  
}
```

Using a cornerRadius of 8 and a continuous corner style is the best I could get to match the existing gray background.

DisclosureGroup



This SwiftUI chapter is locked in this
preview.

The disclosure group gives you the ability to expand and collapse views below it. You supply it with child views that you want collapsed/expanded.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Form

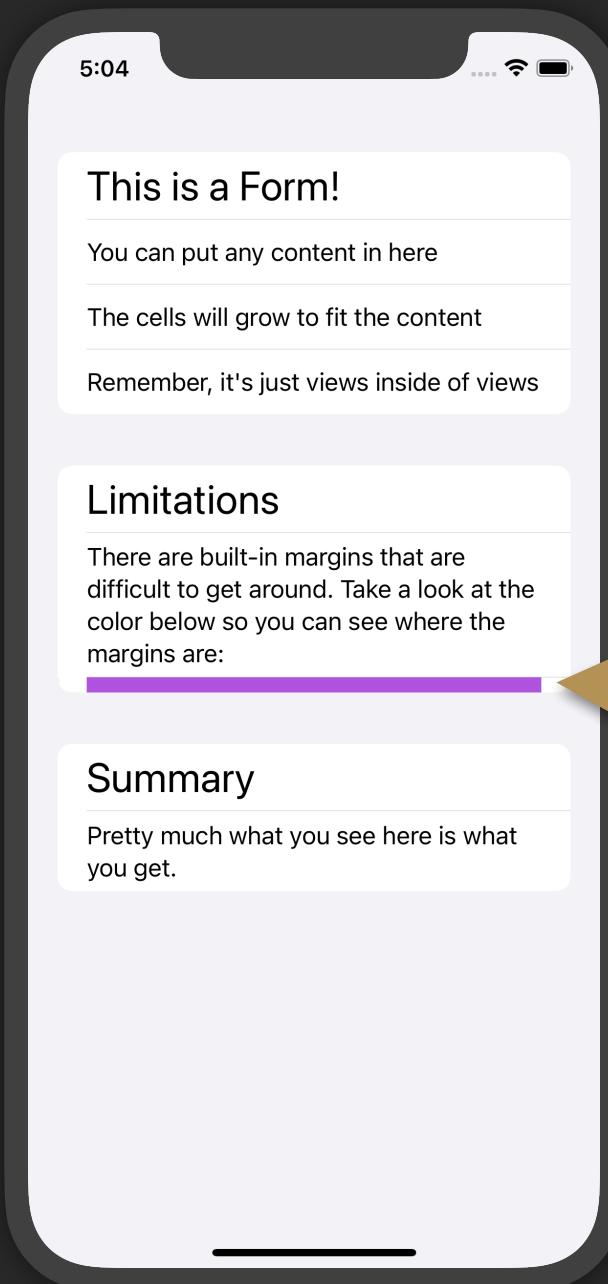


The Form view is a great choice when you want to show settings, options, or get some user input. It is easy to set up and customize as you will see on the following pages.

This is a push-out view.

Introduction

```
struct Form_Intro : View {  
    var body: some View {  
        Form {  
            Section {  
                Text("This is a Form!")  
                    .font(.title)  
                Text("You can put any content in here")  
                Text("The cells will grow to fit the content")  
                Text("Remember, it's just views inside of views")  
            }  
  
            Section {  
                Text("Limitations")  
                    .font(.title)  
                Text("There are built-in margins that are difficult to get around. Take a look at the color below so you can see where the margins are:")  
                Color.purple  
            }  
  
            Section {  
                Text("Summary")  
                    .font(.title)  
                Text("Pretty much what you see here is what you get.")  
            }  
        }  
    }  
}
```

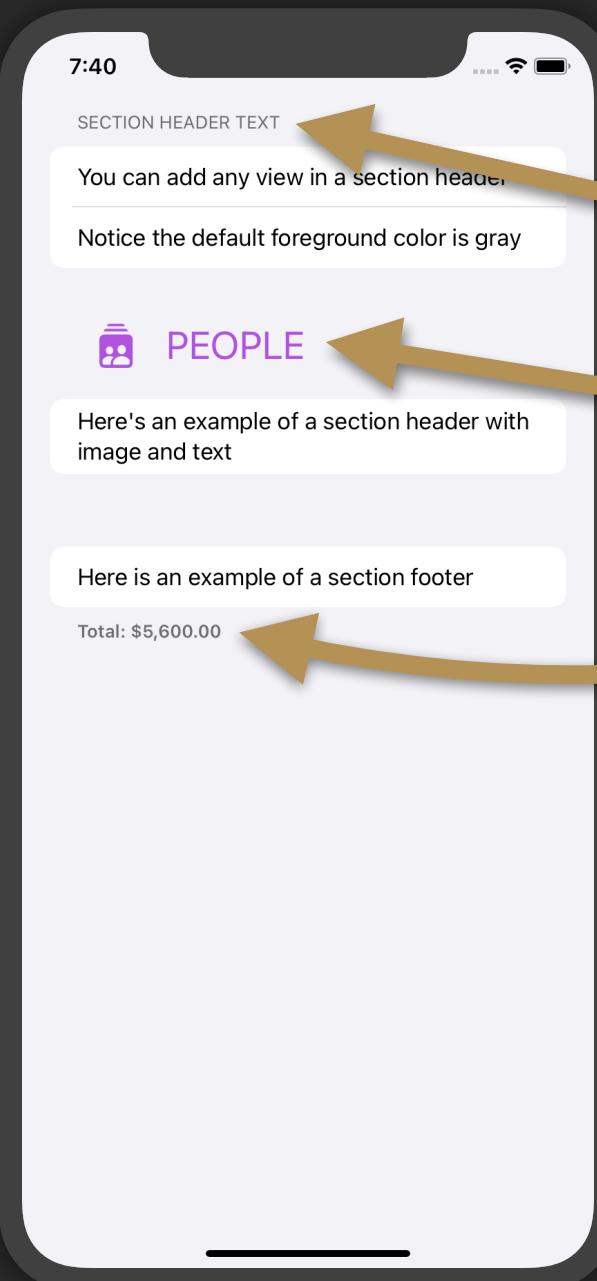


Forms come with a built-in scroll view if the contents exceed the height of the screen.



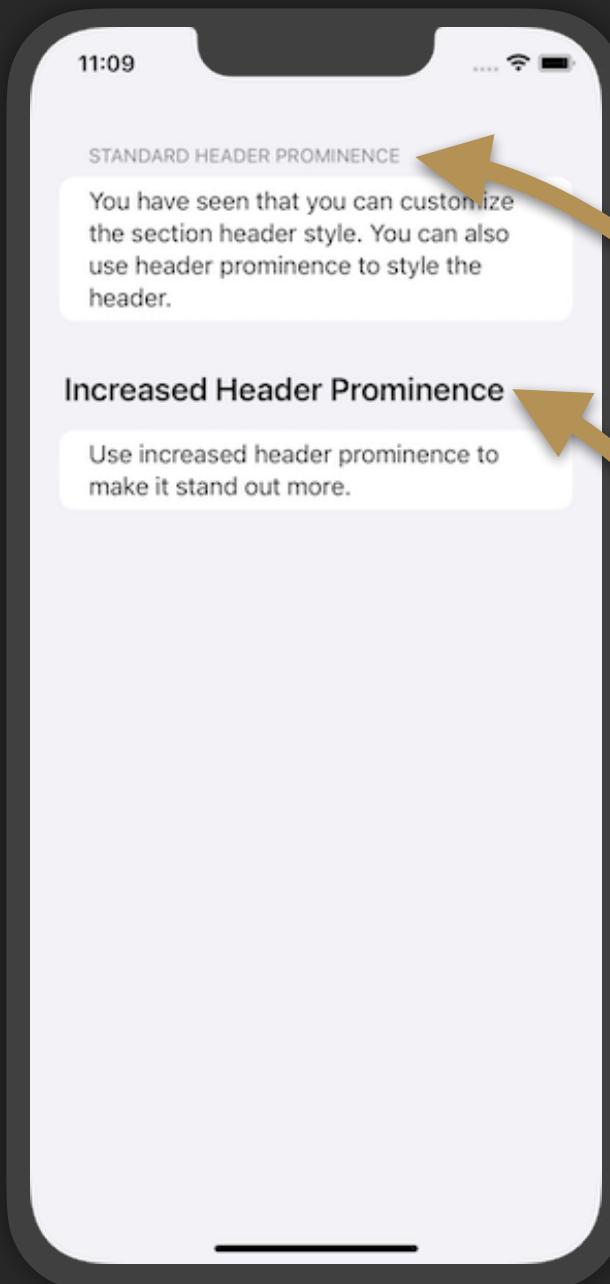
Section Headers and Footers

```
struct Form_HeadersAndFooters : View {  
    var body: some View {  
        Form {  
            Section {  
                Text("You can add any view in a section header")  
                Text("Notice the default foreground color is gray")  
            } header: {  
                Text("Section Header Text")  
            }  
            Section {  
                Text("Here's an example of a section header with image and text")  
            } header: {  
                SectionTextAndImage(name: "People", image: "person.2.square.stack.fill")  
            }  
            Section {  
                Text("Here is an example of a section footer")  
            } footer: {  
                Text("Total: $5,600.00").bold()  
            }  
        }  
    }  
  
    struct SectionTextAndImage: View {  
        var name: String  
        var image: String  
        var body: some View {  
            HStack {  
                Image(systemName: image).padding(.trailing)  
                Text(name)  
            }  
            .padding()  
            .font(.title)  
            .foregroundColor(Color.purple)  
        }  
    }  
}
```





Header Prominence

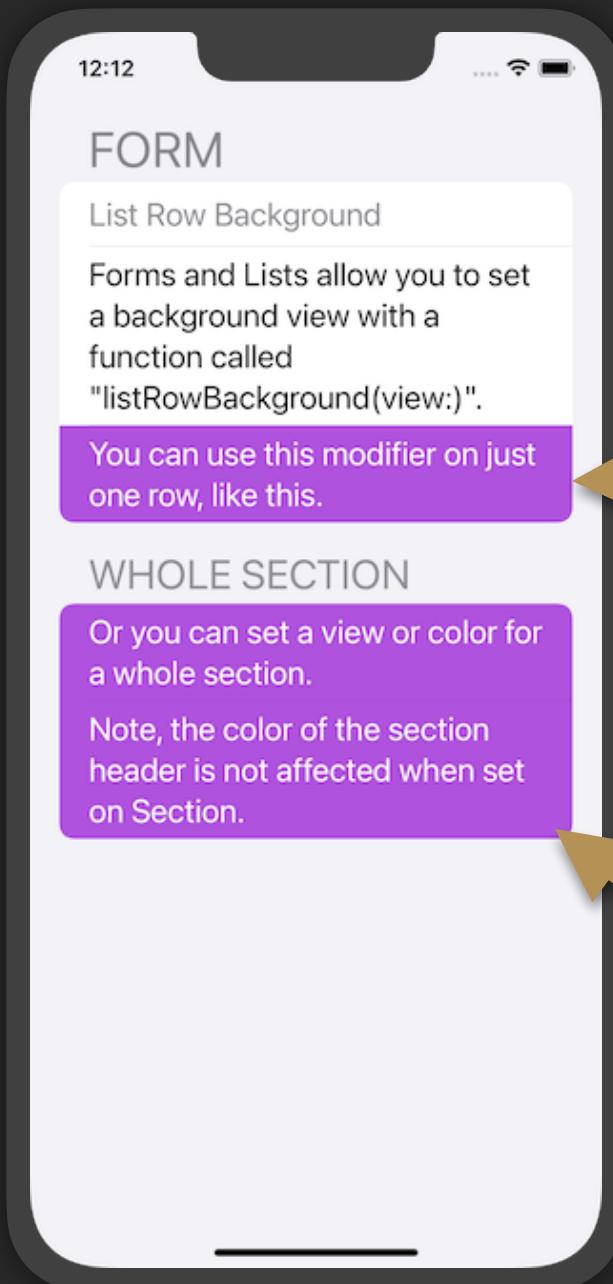


```
struct Form_HeaderProminence: View {  
    var body: some View {  
        Form {  
            Section {  
                Text("You have seen that you can customize the section header style. You can  
also use header prominence to style the header.")  
            } header: {  
                Text("Standard Header Prominence")  
            } .headerProminence(.standard)  
  
            Section {  
                Text("Use increased header prominence to make it stand out more.")  
            } header: {  
                Text("Increased Header Prominence")  
            } .headerProminence(.increased)  
        }  
    }  
}
```

Note: I have found that I can put this modifier on the Section or the Text inside the header closure for it to work.

List Row Background

```
struct Form_ListRowBackground : View {  
    var body: some View {  
        Form {  
            Section {  
                Text("List Row Background")  
                    .foregroundColor(.gray)  
  
                Text("Forms and Lists allow you to set a background view with a function called  
\"listRowBackground(view:)\"")  
  
                Text("You can use this modifier on just one row, like this.")  
                    .listRowBackground(Color.purple)  
                    .foregroundColor(.white)  
            } header: {  
                Text("Form").font(.largeTitle)  
            }  
  
            Section {  
                Text("Or you can set a view or color for a whole section.")  
  
                Text("Note, the color of the section header is not affected when set on  
Section.")  
                    .fixedSize(horizontal: false, vertical: true)  
            } header: {  
                Text("Whole Section")  
                    .font(.title).foregroundColor(.gray)  
            }  
            .foregroundColor(.white)  
            .listRowBackground(Color.purple)  
        }  
        .font(.title2)  
    }  
}
```



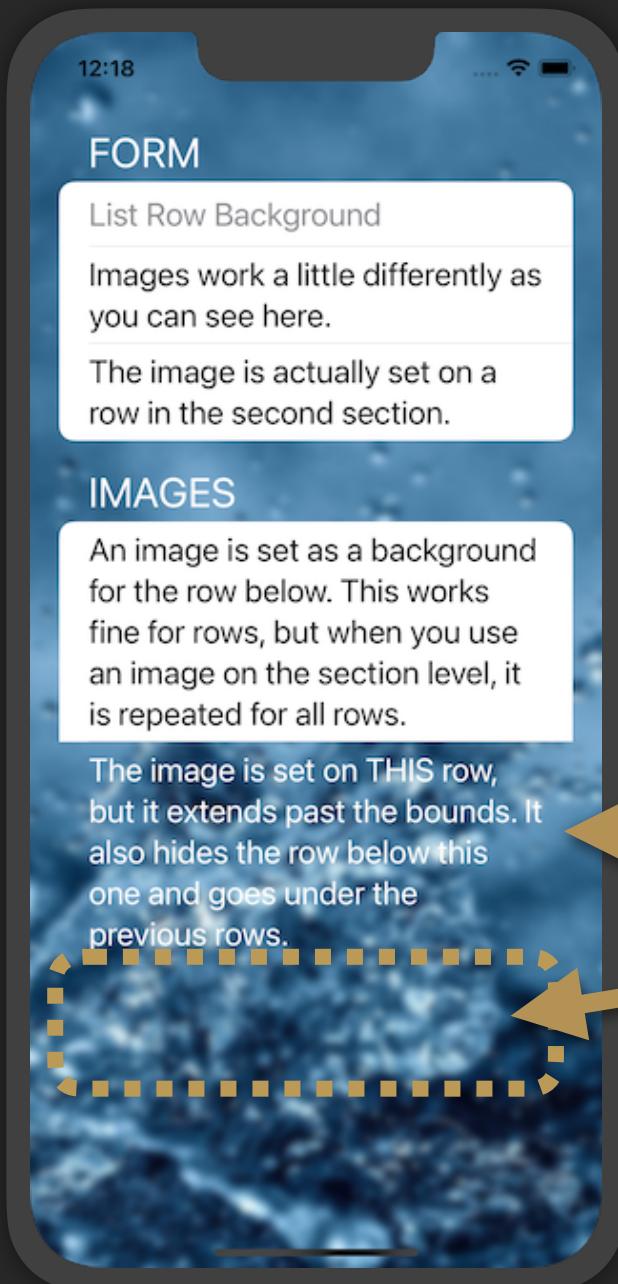


Background Images

```
struct Form_RowBackgroundImage : View {
    var body: some View {
        Form {
            Section {
                Text("List Row Background")
                    .foregroundColor(.gray)
                Text("Images work a little differently as you can see here.")
                Text("The image is actually set on a row in the second section.")
            } header: {
                Text("Form")
                    .font(.title)
                    .foregroundColor(.white)
            }

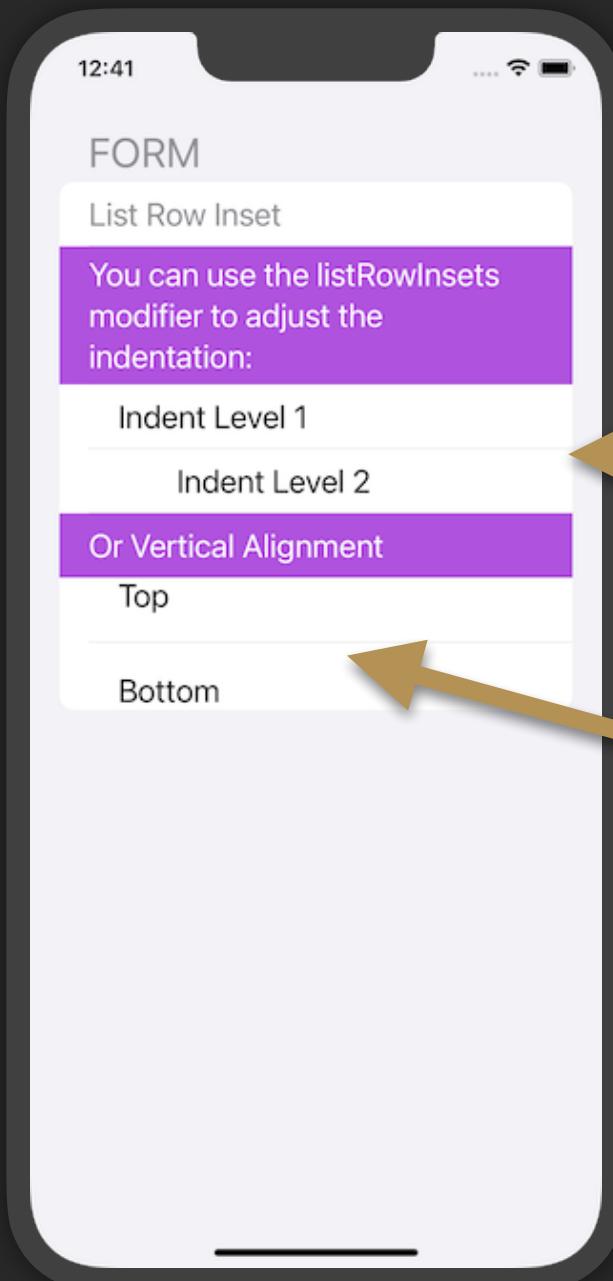
            Section {
                Text("An image is set as a background for the row below. This works fine for rows, but when you use an image on the section level, it is repeated for all rows.")
                Text("The image is set on THIS row, but it extends past the bounds. It also hides the row below this one and goes under the previous rows.")
                    .listRowBackground(Image("water")
                        .blur(radius: 3))
                Text("This row cannot be seen.")
            } header: {
                Text("Images")
                    .font(.title)
                    .foregroundColor(.white)
            }
        }
        .font(.title2)
    }
}
```

Be careful when using images with listRowBackground as they can go into other rows and cover rows.



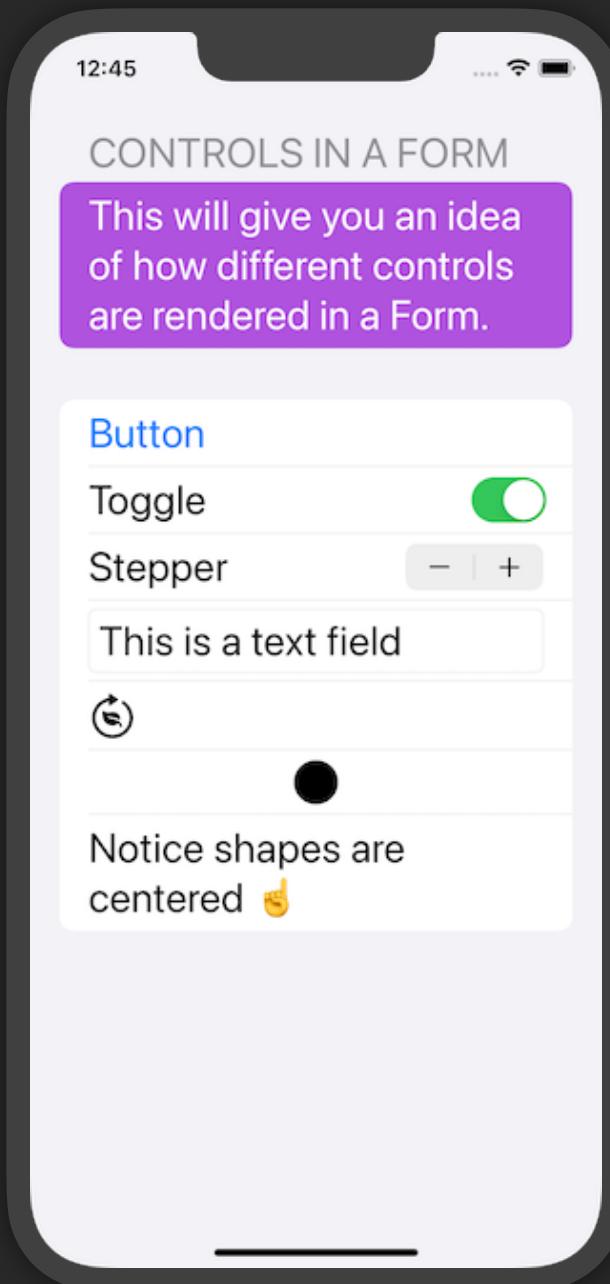
List Row Inset

```
struct Form_ListRowInset : View {  
    var body: some View {  
        Form {  
            Section {  
                Text("List Row Inset")  
                    .foregroundColor(.gray)  
                Text("You can use the listRowInsets modifier to adjust the indentation:")  
                    .foregroundColor(.white)  
                    .listRowBackground(Color.purple)  
            }  
        }  
    }  
}
```



```
Text("Indent Level 1")  
    .listRowInsets(EdgeInsets(top: 0, leading: 40, bottom: 0, trailing: 0))  
  
Text("Indent Level 2")  
    .listRowInsets(EdgeInsets(top: 0, leading: 80, bottom: 0, trailing: 0))  
  
Text("Or Vertical Alignment")  
    .foregroundColor(.white)  
    .listRowBackground(Color.purple)  
  
Text("Top")  
    .listRowInsets(EdgeInsets(top: -20, leading: 40, bottom: 0, trailing: 0))  
  
Text("Bottom")  
    .listRowInsets(EdgeInsets(top: 20, leading: 40, bottom: 0, trailing: 0))  
}  
header: {  
    Text("Form")  
        .font(.title)  
        .foregroundColor(.gray)  
}  
}  
.  
font(.title2)  
}
```

With Controls

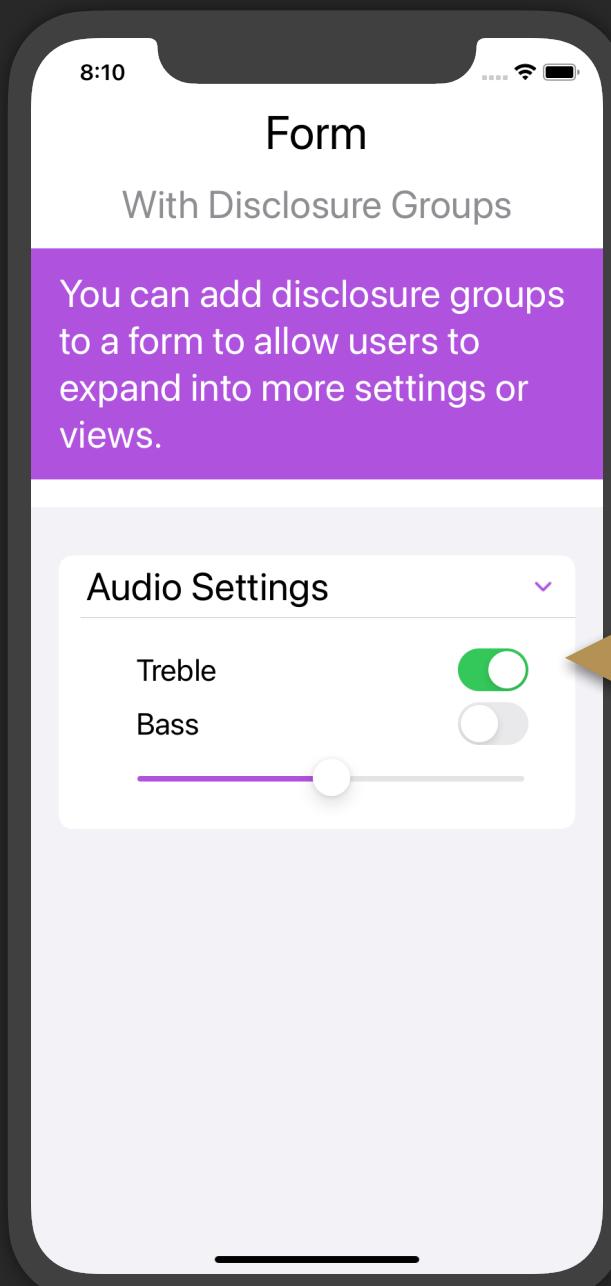


```
struct Form_WithControls : View {
    @State private var isOn = true
    @State private var textFieldData = "This is a text field"

    var body: some View {
        Form {
            Section {
                Text("This will give you an idea of how different controls are rendered in a Form.")
                    .foregroundColor(.white)
                    .listRowBackground(Color.purple)
            } header: {
                Text("Controls in a form")
                    .font(.title)
                    .foregroundColor(Color.gray)
            }
            Section {
                Button(action: {}) { Text("Button") }
                Toggle(isOn: $isOn) { Text("Toggle") }
                Stepper(onIncrement: {}, onDecrement: {}) { Text("Stepper") }
                TextField("", text: $textFieldData)
                    .textFieldStyle(.roundedBorder)
                Image(systemName: "leaf.arrow.circlepath").font(.title)
                Circle()
                    .frame(height: 30)
                Text("Notice shapes are centered 🤝")
            }
        }
        .font(.title)
    }
}
```



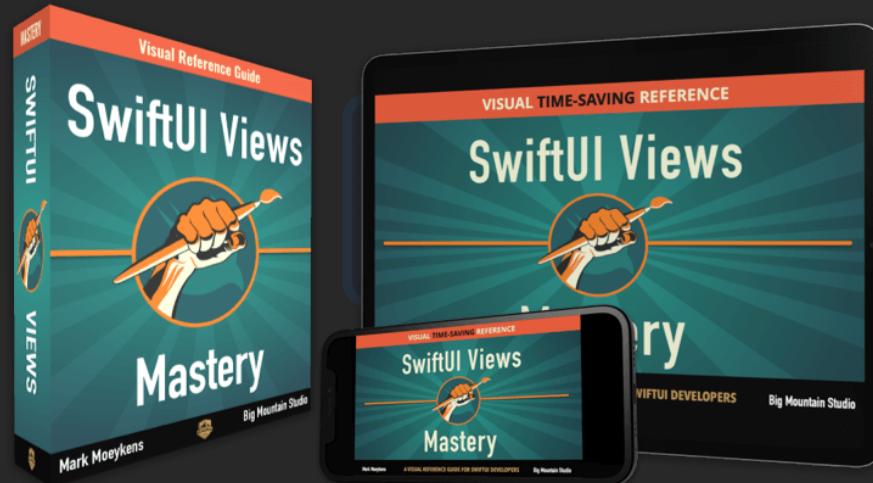
With the Disclosure Group



```
struct Form_WithDisclosureGroup: View {  
    @State private var settingsExpanded = true  
    @State private var trebleOn = true  
    @State private var bassOn = false  
    @State private var levels = 0.5  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Form",  
                subtitle: "With Disclosure Groups",  
                desc: "You can add disclosure groups to a form to allow users to expand  
into more settings or views.",  
                back: .purple, textColor: .white)  
  
            Form {  
                DisclosureGroup("Audio Settings", isExpanded: $settingsExpanded) {  
                    VStack {  
                        Toggle("Treble", isOn: $trebleOn)  
                        Toggle("Bass", isOn: $bassOn)  
                        Slider(value: $levels)  
                    }  
                    .font(.title2)  
                    .padding()  
                }  
            }  
            .font(.title)  
            .accentColor(.purple)  
        }  
    }  
}
```

See [Control Views > DisclosureGroup](#) for more info.

Gauge



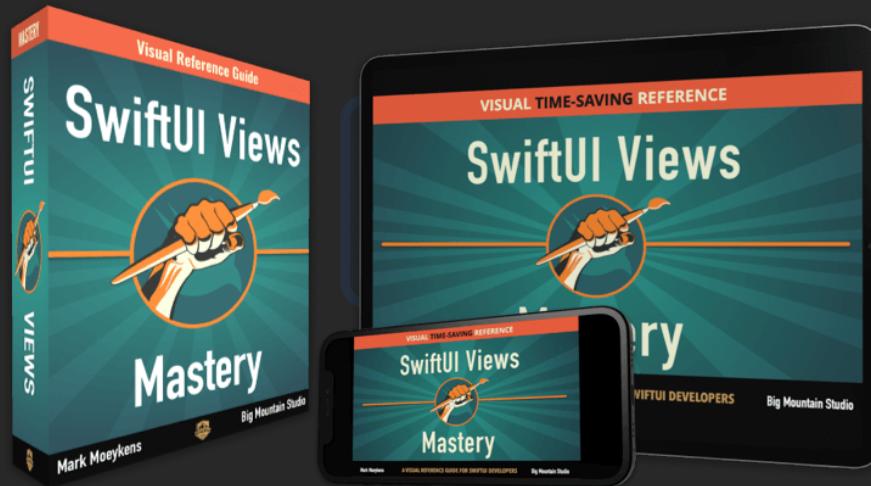
This SwiftUI chapter is locked in this
preview.

Use the gauge to show a value in a range. The value and range can be shown in a linear or circular way.

When the gauge is shown in a linear way.
When the gauge is shown in a circular way.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

GroupBox



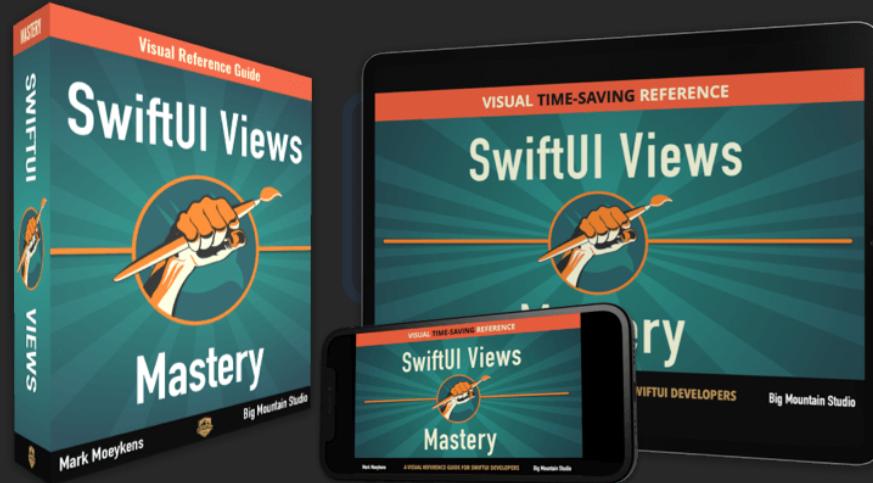
This SwiftUI chapter is locked in this
preview.

The group box is container for grouping similar items together.

The GroupBox is a pull-in

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Label



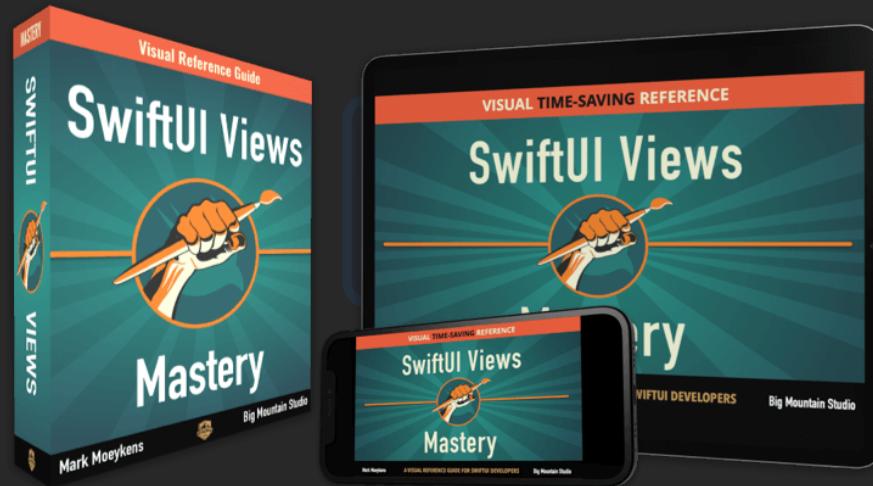
This SwiftUI chapter is locked in this

The label view is a pretty simple view that will handle the layout, spacing and formatting of an image and text that you pass into it.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

LabeledContent



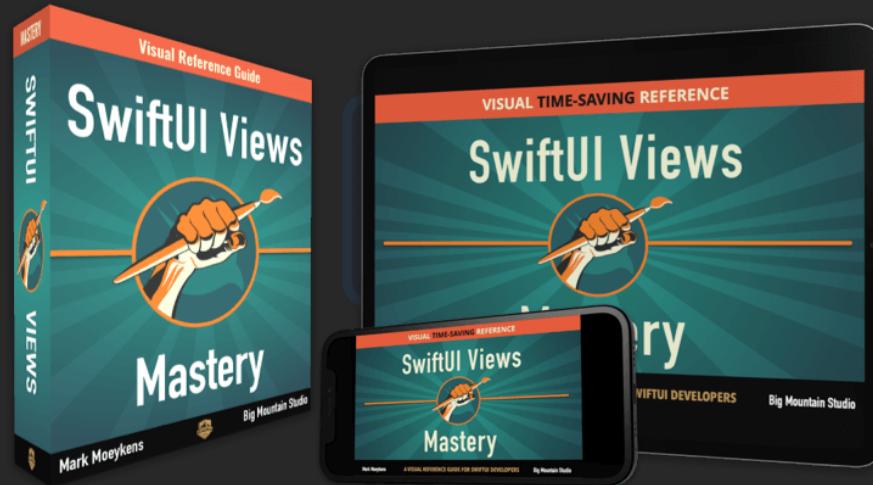
This SwiftUI chapter is locked in this
preview.

You may have noticed some controls have this pattern of a label on the left and the control on the right. Think of this view as a template where you can supply a label on the left and your own content on the right.

This view pushes out horizontal

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Link



This SwiftUI chapter is locked in this
preview.

The Link is similar to the Button or the NavigationLink except it can navigate you to a place outside your app.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

List

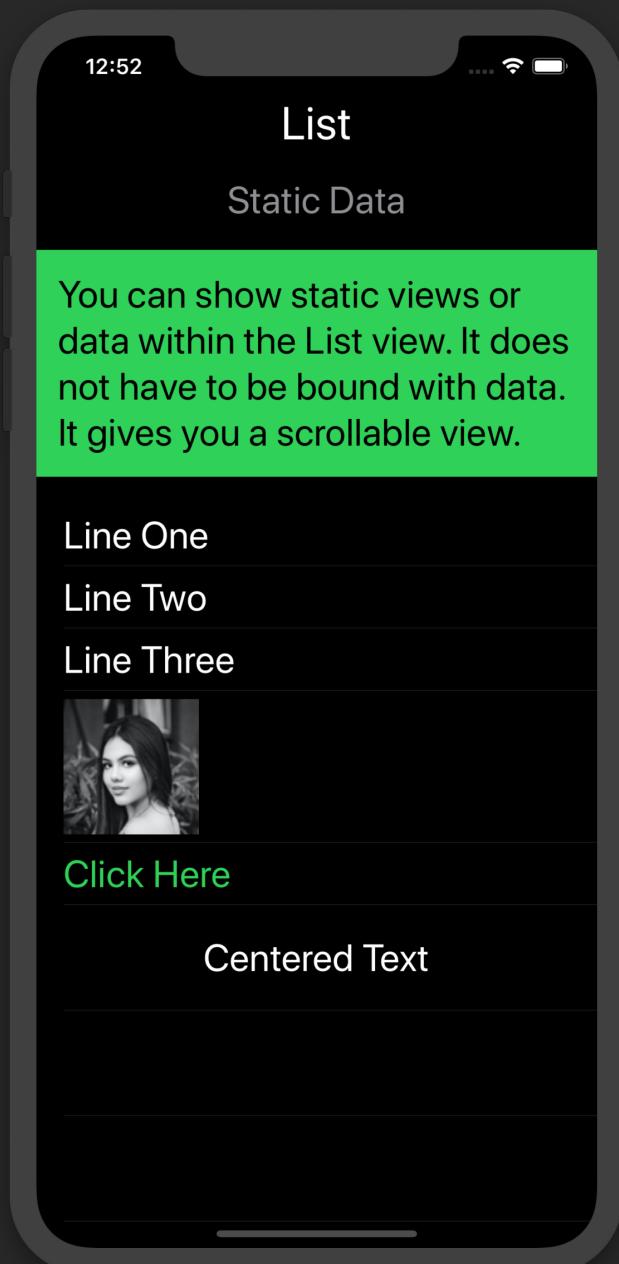


Using a List view is the most efficient way of displaying vertically scrolling data. You can display data in a ScrollView, as you will see later on, but it will not be as efficient in terms of memory or performance as the List view.

With Static Views

```
struct List_WithStaticData: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("List").font(.largeTitle)  
            Text("Static Data").font(.title).foregroundColor(.gray)  
            Text("You can show static views or data within the List view. It does not have to be bound with data. It gives you a scrollable view.")  
                .frame(maxWidth: .infinity)  
                .font(.title).padding()  
                .background(Color.green)  
                .foregroundColor(.black)  
  
            List {  
                Text("Line One")  
                Text("Line Two")  
                Text("Line Three")  
                Image("profile")  
                Button("Click Here", action: {})  
                    .foregroundColor(.green)  
                HStack {  
                    Spacer()  
                    Text("Centered Text")  
                    Spacer()  
                }.padding()  
            }  
                .font(.title)  
        }  
    }  
}
```

Note: Like other container views, you cannot have more than 10 views inside.



With Data

```
struct List_WithData : View {  
    var stringArray = ["This is the simplest List", "Evans", "Lemuel James Guerrero", "Mark",  
"Durtschi", "Chase", "Adam", "Rodrigo", "Notice the automatic wrapping when the text is longer"]  
  
    var body: some View {  
        List(stringArray, id: \.self) { string in ←  
            Text(string)  
        }  
        .font(.largeTitle) // Apply this font style to all items in the list  
    }  
}
```

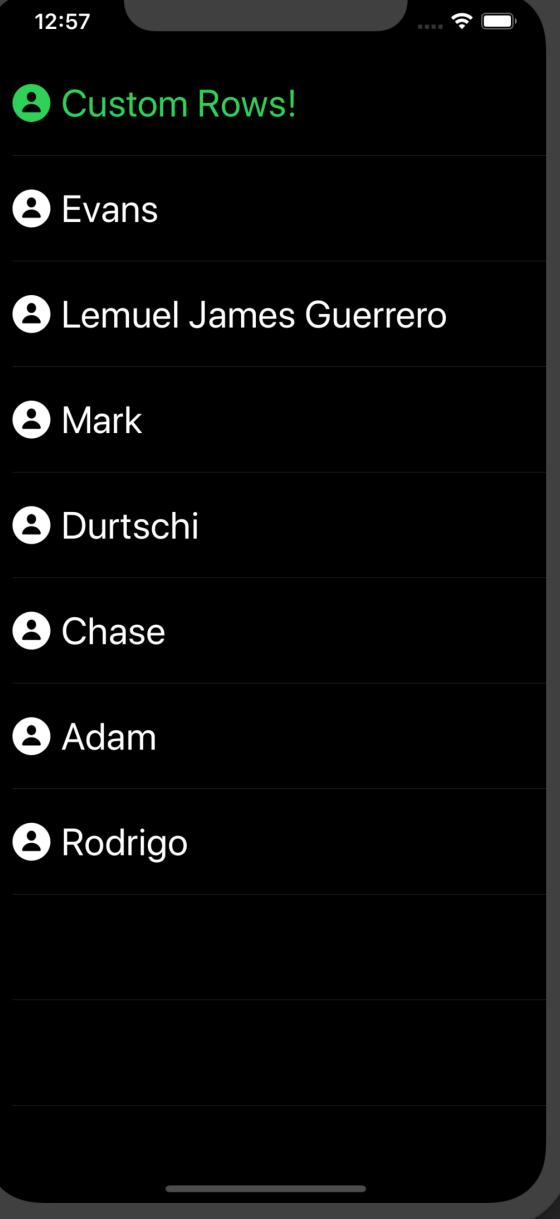
The List view can iterate through an array of data and pass in one item at a time to its closure.

What is that `.id` parameter?

You use this parameter to tell the List how it can uniquely identify each row by which value. The List needs to know this so it can compare rows by this value to perform different operations like reordering and deleting rows for us.

In this scenario, we are using "self" to say, "Just use the value of the string itself to uniquely identify each row."

Custom Rows



```
struct List_CustomRows : View {  
    var data = ["Custom Rows!", "Evans", "Lemuel James Guerrero", "Mark", "Durtschi", "Chase",  
    "Adam", "Rodrigo"]
```

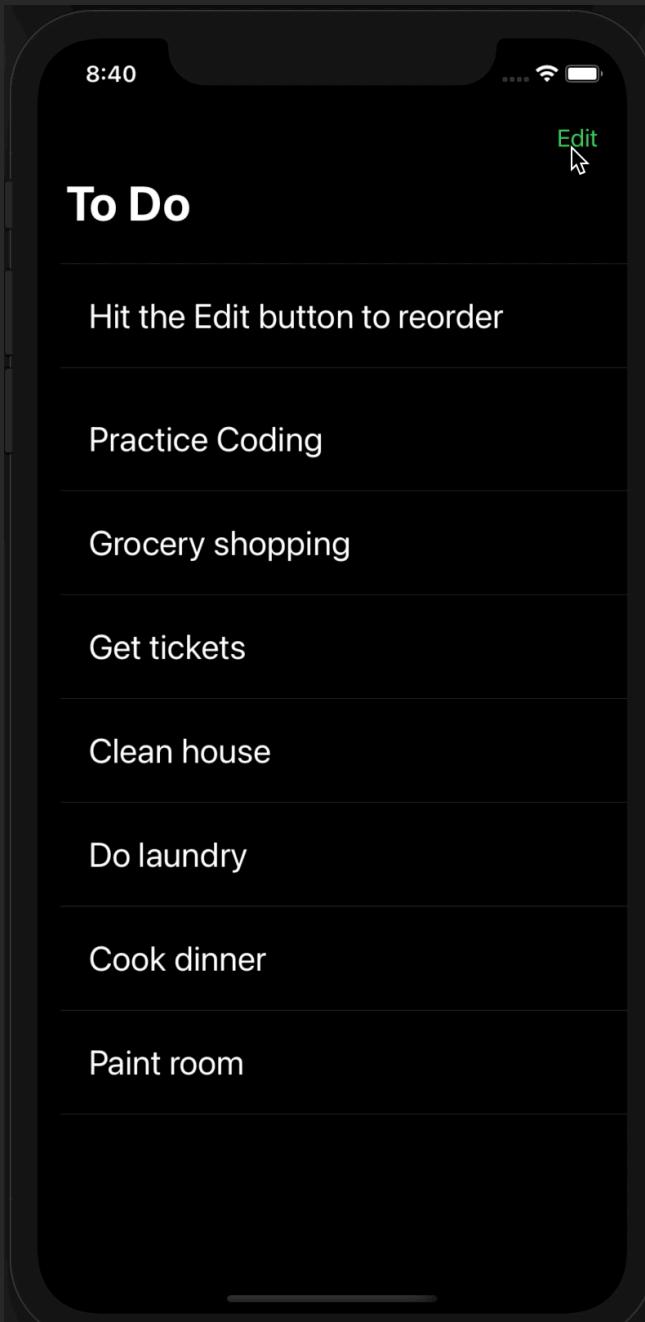
```
        var body: some View {  
            List(data, id: \.self) { datum in  
                CustomRow(content: datum)  
            }  
        }  
}
```



```
struct CustomRow: View {  
    var content: String  
  
    var body: some View {  
        HStack {  
            Image(systemName: "person.circle.fill")  
            Text(content)  
            Spacer()  
        }  
        .foregroundColor(content == "Custom Rows!" ? Color.green : Color.primary)  
        .font(.title)  
        .padding([.top, .bottom])  
    }  
}
```

Extracting rows into separate views is a common practice in SwiftUI. You can then have a separate preview just for the row.

Move Rows



```
struct List_MoveRow : View {
    @State var data = ["Hit the Edit button to reorder", "Practice Coding", "Grocery shopping",
    "Get tickets", "Clean house", "Do laundry", "Cook dinner", "Paint room"]
```

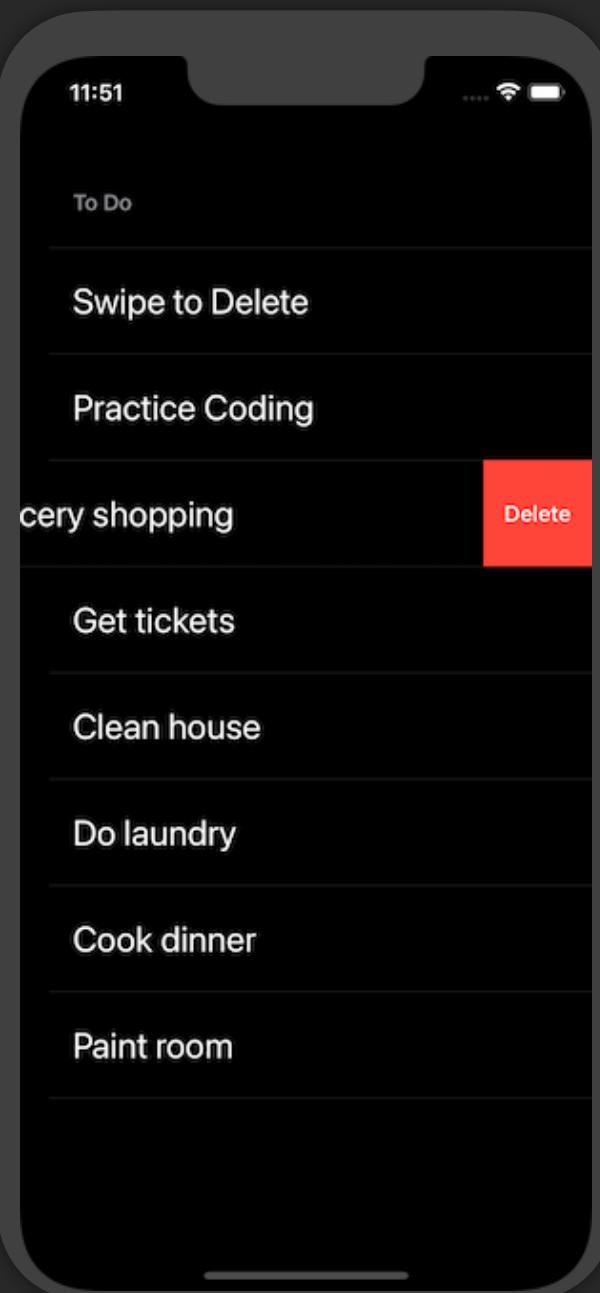
```
var body: some View {
    NavigationView {
        List {
            ForEach(data, id: \.self) { datum in
                Text(datum).font(Font.system(size: 24)).padding()
            }
            .onMove { source, destination in
                data.move(fromOffsets: source, toOffset: destination)
            }
        }
        .navigationTitle("To Do")
        .toolbar {
            ToolbarItem { EditButton() }
        }
        .tint(.green) // Changes color of buttons
    }
}
```

The `onMove` modifier goes on the `ForEach`, not the `List`.

What is `EditButton()`?

This is a built-in function that returns a view (Button) that will automatically toggle edit mode on the List. Its text says "Edit" and then when tapped you will see the move handles appear on the rows and the button text says "Done".

Delete Rows



```
struct List_Delete : View {
    @State var data = ["Swipe to Delete", "Practice Coding", "Grocery shopping", "Get tickets",
    "Clean house", "Do laundry", "Cook dinner", "Paint room"]

    var body: some View {
        List {
            Section {
                ForEach(data, id: \.self) { datum in
                    Text(datum).font(Font.system(size: 24)).padding()
                }
                .onDelete { offsets in
                    data.remove(atOffsets: offsets)
                }
            } header: {
                Text("To Do")
                .padding()
            }
        }
        .listStyle(.plain)
    }
}
```

onDelete, onMove, onInsert

These three functions only work on views that implement the DynamicViewContent protocol. Currently, the only view that conforms to the DynamicViewContent protocol is the ForEach view. So these functions are only available on a ForEach view, not a List view.

Selecting a Row

```
struct List_Selection_Single: View {  
    @State private var data = ["Practice Coding", "Grocery shopping", "Get tickets",  
        "Clean house", "Do laundry", "Cook dinner", "Paint room"]  
    @State private var selection: String?  
}
```

Use an optional type to store which row is selected.

```
var body: some View {  
    NavigationView {  
        VStack(spacing: 0) {  
            HeaderView("",  
                subtitle: "Selecting a Row",  
                desc: "Use the selection parameter in List and supply a @State property to hold the selection.")  
        }  
    }
```



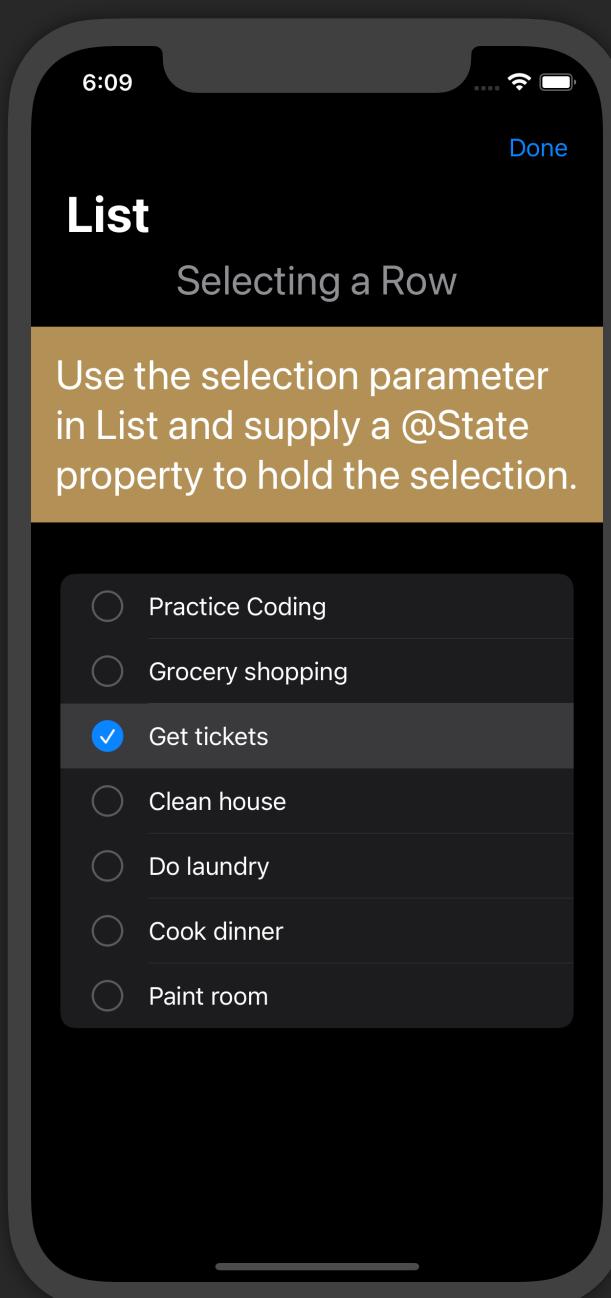
```
        List(data, id: \.self, selection: $selection) { item in  
            Text(item)  
        }  
        .font(.title)  
        .navigationTitle("List")  
        .toolbar { EditButton() }  
    }  
}
```

Bind the selection parameter to your @State property above using the dollar sign (\$).



```
}  
}
```

You need the edit button to enable edit mode for row selection.

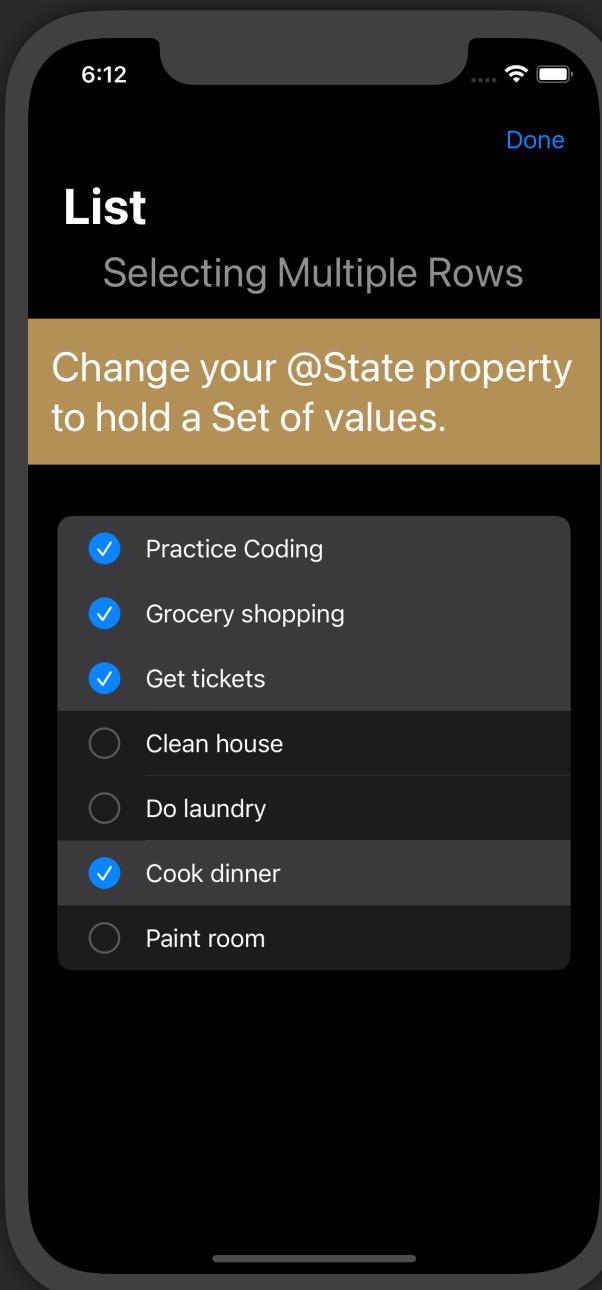




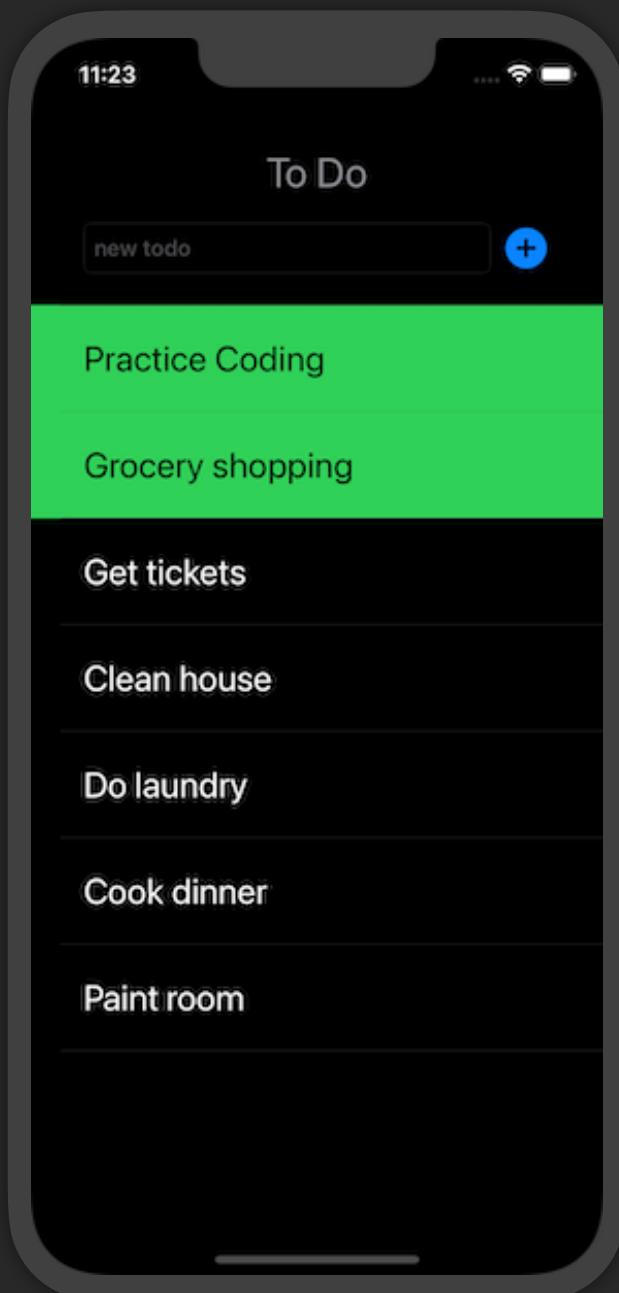
Selecting Multiple Rows

```
struct List_Selection_Multiple: View {  
    @State private var data = ["Practice Coding", "Grocery shopping", "Get tickets",  
                             "Clean house", "Do laundry", "Cook dinner", "Paint room"]  
    @State private var selections = Set<String>()  
  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 0) {  
                HeaderView("",  
                          subtitle: "Selecting Multiple Rows",  
                          desc: "Change your @State property to hold a Set of values.")  
  
                List(data, id: \.self, selection: $selections) { item in  
                    Text(item)  
                }  
                .font(.title)  
                .navigationTitle("List")  
                .toolbar { EditButton() }  
            }  
        }  
    }  
}
```

By changing the type to a Set, the List will automatically know it can hold multiple selection values.



List Row Background



```

struct Todo: Identifiable {
    let id = UUID()
    var action = ""
    var due = ""
    var isIndented = false
}

struct List_ListRowBackground : View {
    @State private var newToDo = ""

    @State var data = [
        Todo(action: "Practice Coding", due: "Today"),
        Todo(action: "Grocery shopping", due: "Today"),
        Todo(action: "Get tickets", due: "Tomorrow"),
        Todo(action: "Clean house", due: "Next Week"),
        Todo(action: "Do laundry", due: "Next Week"),
        Todo(action: "Cook dinner", due: "Next Week"),
        Todo(action: "Paint room", due: "Next Week")
    ]

    var body: some View {
        List {
            Section {
                ForEach(data) { datum in
                    Text(datum.action)
                        .font(Font.system(size: 24))
                        .foregroundColor(self.getTextColor(due: datum.due))
                    // Turn row green if due today
                    .listRowBackground(datum.due == "Today" ? Color.green : Color.clear)
                    .padding()
                }
            }
        }
    }
}

```

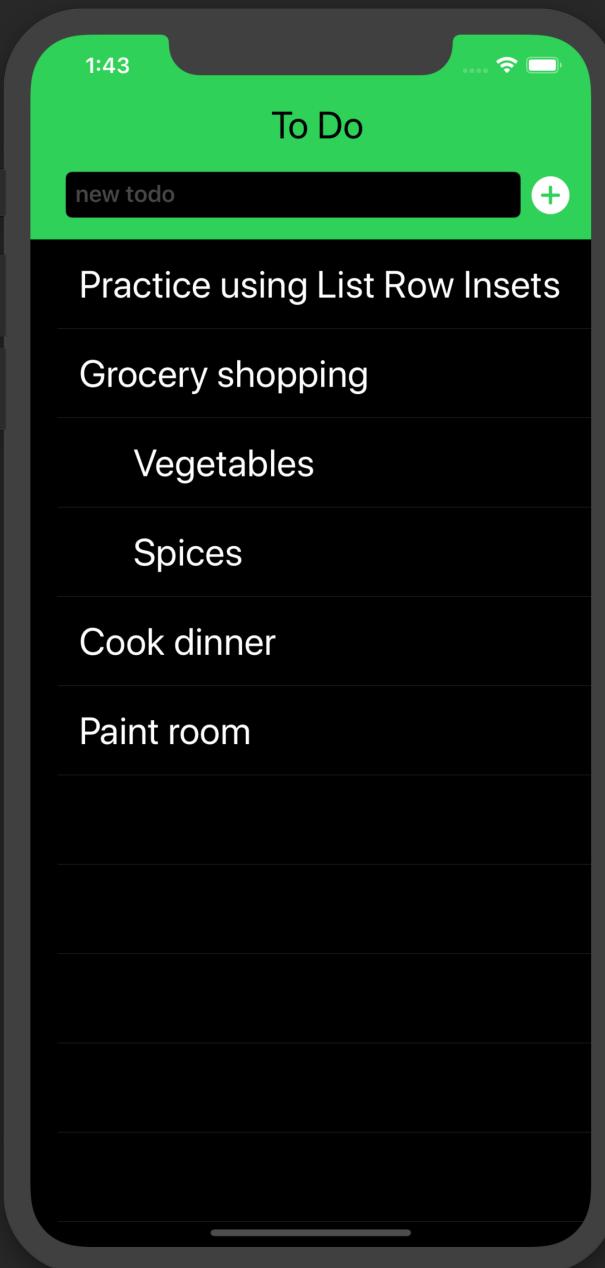
Notice the `.listRowBackground` function is on the view inside the `ForEach`. You want to call this function on whatever view will be inside the row, not on the List itself.

List Row Background Continued

```
    } header: {
        VStack {
            Text("To Do").font(.title)
            HStack {
                TextField("new todo", text: $newToDo)
                    .textFieldStyle(.roundedBorder)
                Button(action: {
                    data.append(Todo(action: newToDo))
                    newToDo = ""
                }) {
                    Image(systemName: "plus.circle.fill").font(.title)
                }
            }
            .padding(.bottom)
        }
        .listStyle(.plain)
    }

    // This logic was inline but the compiler said it was "too complex" 🤦
    private func getTextColor(due: String) -> Color {
        due == "Today" ? Color.black : Color.primary
    }
}
```

List Row Inset



```
struct List_ListRowInsets : View {  
    @State private var newToDo = ""  
  
    @State var data = [  
        Todo(action: "Practice using List Row Insets", due: "Today"),  
        Todo(action: "Grocery shopping", due: "Today"),  
        Todo(action: "Vegetables", due: "Today", isIndented: true),  
        Todo(action: "Spices", due: "Today", isIndented: true),  
        Todo(action: "Cook dinner", due: "Next Week"),  
        Todo(action: "Paint room", due: "Next Week")  
    ]  
  
    var body: some View {  
        VStack {  
            VStack {  
                Text("To Do")  
                    .font(.title)  
                    .foregroundColor(.black)  
                HStack {  
                    TextField("new todo", text: $newToDo)  
                        .textFieldStyle(.roundedBorder)  
                    Button(action: {  
                        data.append(Todo(action: newToDo))  
                        newToDo = ""  
                    }) {  
                        Image(systemName: "plus.circle.fill")  
                            .font(.title)  
                    }  
                }  
            }  
        }  
    }  
}
```

See next page for the code
that insets the rows.



List Row Inset Continued

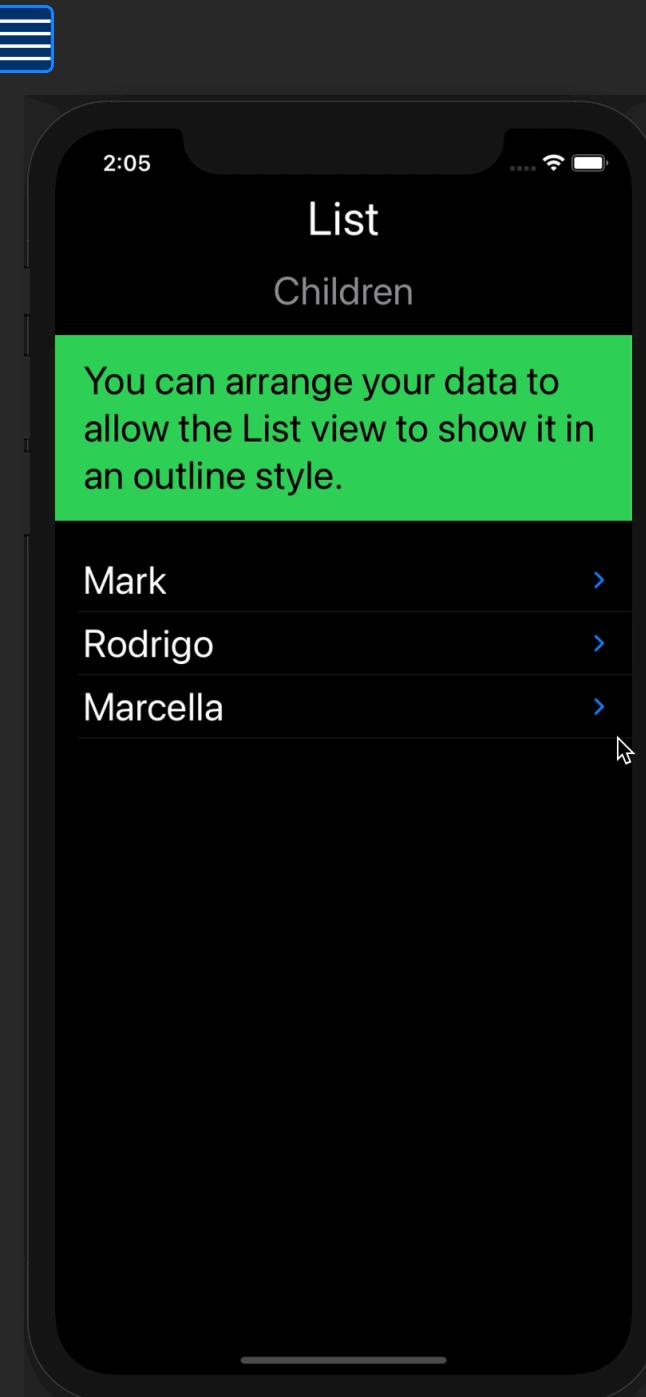
```
        .foregroundColor(.white)
    }
}

.padding()
.background(Color.green)

List {
    ForEach(self.data) { datum in
        Text(datum.action)
            .font(.title)
            .padding()
        // Inset row based on data
        .listRowInsets(EdgeInsets(top: 0,
                                  leading: datum.isIndented ? 60 : 20,
                                  bottom: 0, trailing: 0))
    }
    .listStyle(.plain)
}
}
```

I'm using a condition here to determine just how much to inset the row.

List With Children



```
// Need to conform to Identifiable
struct Parent: Identifiable {
    var id = UUID()
    var name = ""
    var children: [Parent]? // Had to make this optional
}

struct List_WithChildren: View {
    var parents = [Parent(name: "Mark",
                          children: [Parent(name: "Paola")]),
                  Parent(name: "Rodrigo",
                          children: [Parent(name: "Kai"), Parent(name: "Brennan"),
                                    Parent(name: "Easton")]),
                  Parent(name: "Marcella",
                          children: [Parent(name: "Sam"), Parent(name: "Melissa"),
                                    Parent(name: "Melanie")])]

    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("List",
                       subtitle: "Children",
                       desc: "You can arrange your data to allow the List view to show it in an
                           outline style.", back: .green)

            List(parents, children: \.children) { parent in
                Text("\(parent.name)")
            }
            .font(.title)
        }
    }
}
```

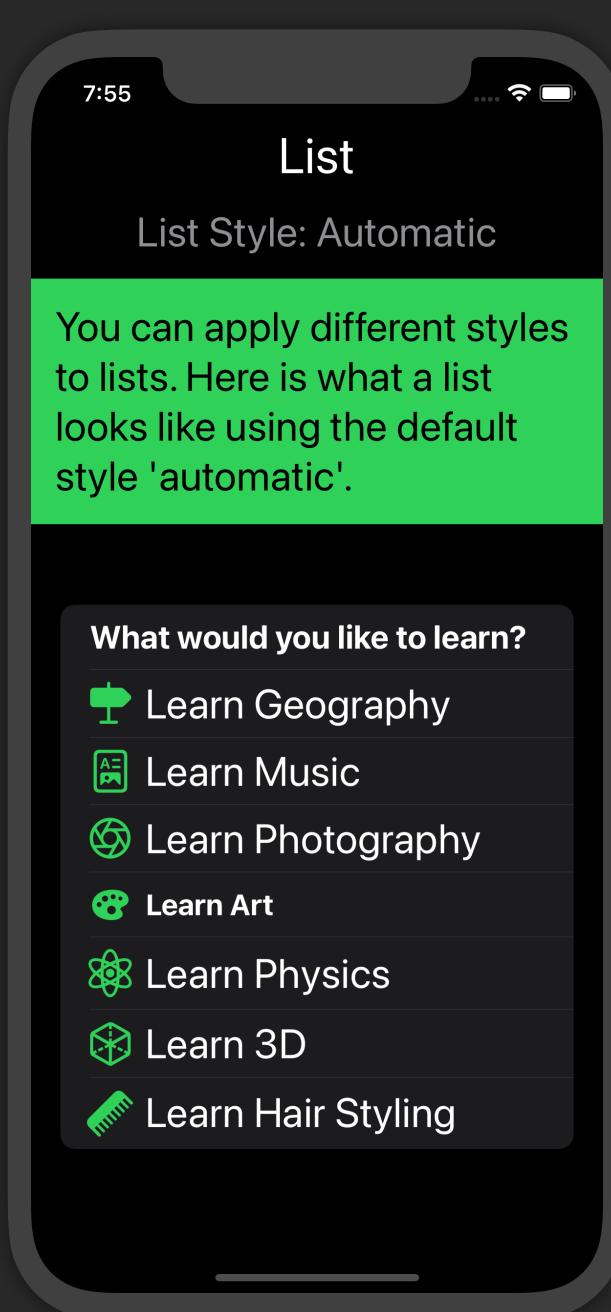
If you have nested data, this could be a good way to represent it in a List.

Use the List init with the children parameter and use a key path to point to your nested property.

ListStyle: Automatic

```
struct List_ListStyle_Automatic: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("List",  
                subtitle: "List Style: Automatic",  
                desc: "You can apply different styles  
                like using the default style 'automatic'.",  
                back: .green)  
            List {  
                Text("What would you like to learn?")  
                    .font(.title2)  
                    .fontWeight(.bold)  
                Label("Learn Geography", systemImage: "signpost.right.fill")  
                Label("Learn Music", systemImage: "doc.richtext")  
                Label("Learn Photography", systemImage: "camera.aperture")  
                Label("Learn Art", systemImage: "paintpalette.fill")  
                    .font(Font.system(.title3).weight(.bold))  
                Label("Learn Physics", systemImage: "atom")  
                Label("Learn 3D", systemImage: "cube.transparent")  
                Label("Learn Hair Styling", systemImage: "comb.fill")  
            }  
            .accentColor(.green)  
            .listStyle(.automatic)  
        }  
        .font(.title)  
    }  
}
```

Note: You do not have to apply this modifier if the value is automatic. This is the default.



Additional List Content

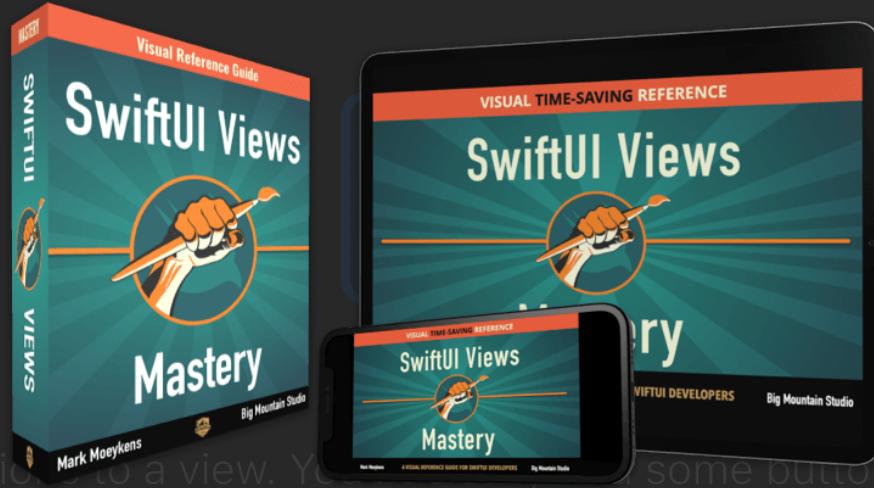
Discover even more you can do with a list including: customizing separators (or removing them), swipe actions, section separators, safe area insets, apply tints and header/footer list styles.



This SwiftUI chapter is locked in this
preview.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Menu



The Menu view allows you to attach actions to a view. You can add some buttons (with or without images) and define a label, or a visible view to the menu. When the user taps the label, the actions will show.



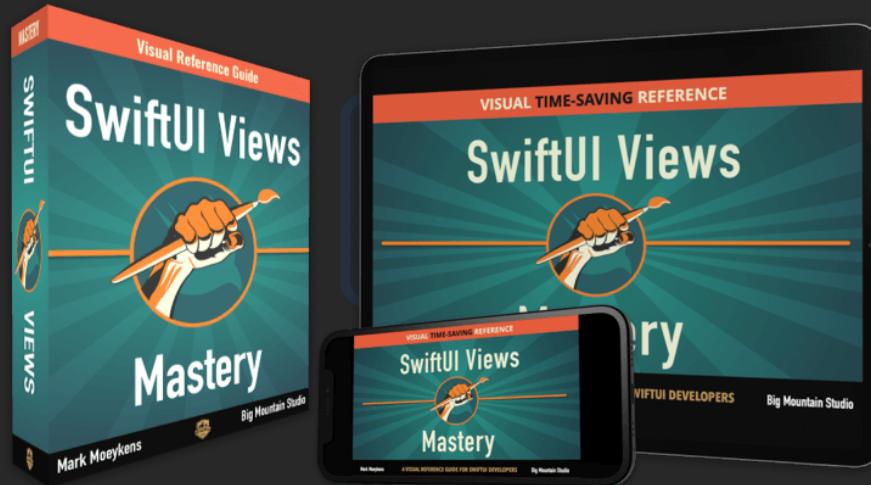
This SwiftUI chapter is locked in this

This is similar to the `contextMenu` modifier (in the Controls Modifiers chapter) where you can attach a menu to any view that becomes visible when you long-press the view.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

MultiDatePicker



This SwiftUI chapter is locked in this preview.

The multi-date picker allows you to select many different days. But an important thing to understand is it does not allow the user to select a range of dates...just individual dates.



The selected dates are not ordered.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

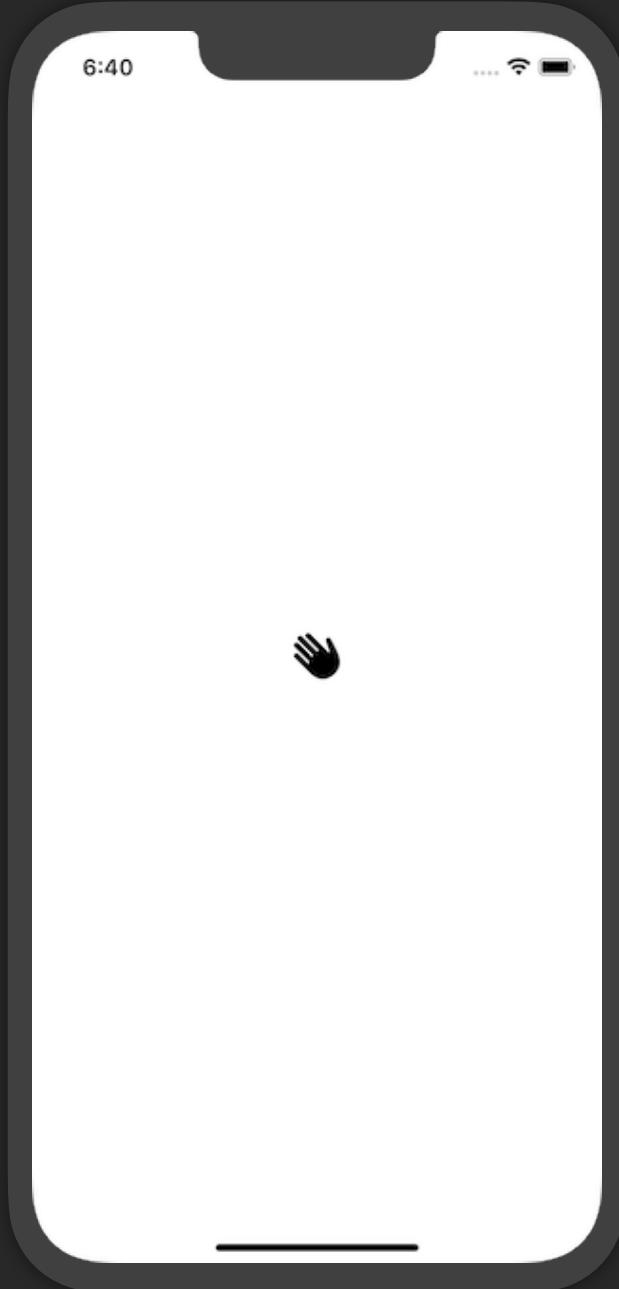
NavigationStack



The NavigationStack is slightly different in that it will fill the whole screen when used. You will never have to specify its size. It is used to help you navigate from one screen to another with built-in ways to navigate forward and back. This chapter will also teach you how to use and customize the navigation stack.

This is a push-out view.

Introduction



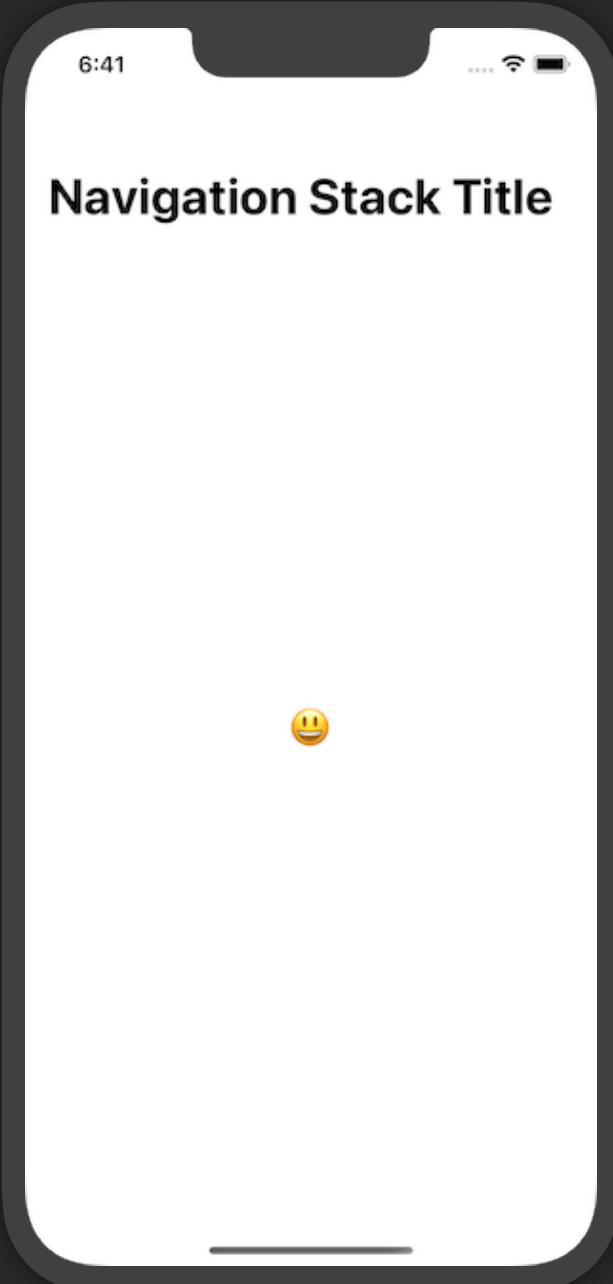
```
struct Navigation_Intro : View {  
    var body: some View {  
        NavigationStack {  
            Image(systemName: "hand.wave.fill")  
                .font(.largeTitle)  
        }  
    }  
}
```

By default, the navigation stack will not show anything if there is no title modifier added for it to use.

The navigation stack bar is there at the top, you just can't see it.

Navigation Title

```
struct Navigation_NavigationTitle: View {  
    var body: some View {  
        NavigationStack {  
            Text("😊")  
                .navigationTitle("Navigation Stack Title")  
        }  
        .font(.title)  
    }  
}
```



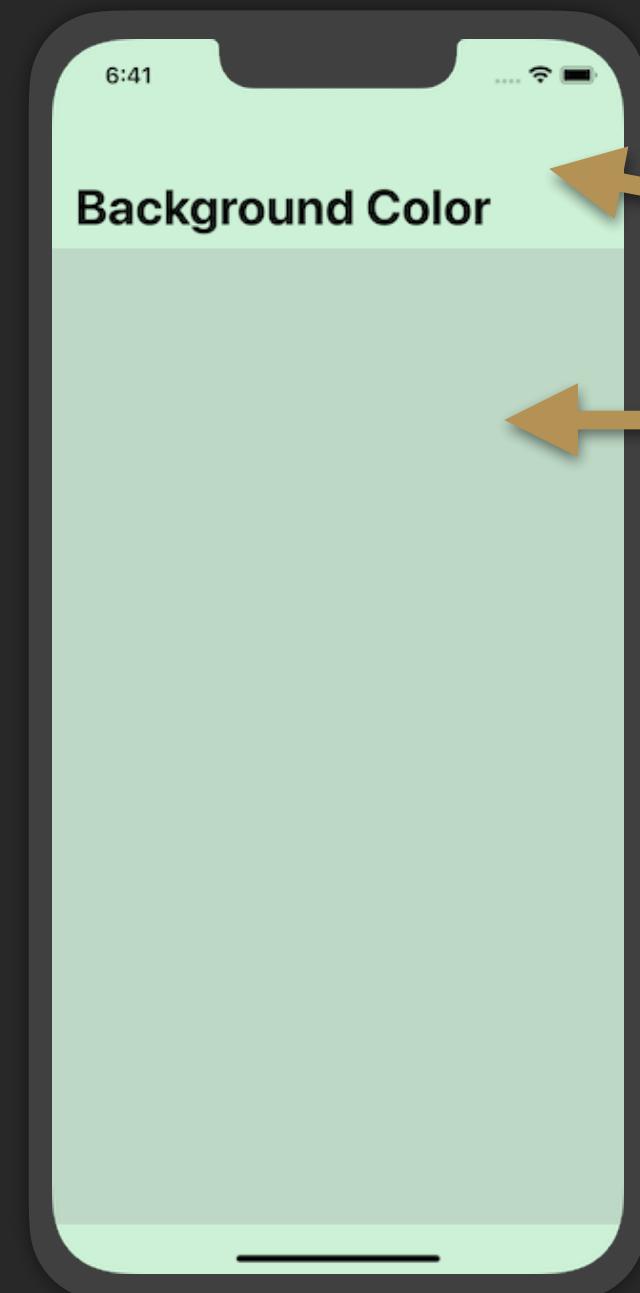
The navigation stack will look for a navigation title modifier INSIDE itself.

By default, a larger navigation bar is used for the navigation stack.



Warning: If you put the navigation title on the navigation stack itself, it will not work.

Background Color



```
struct Navigation_BackgroundColor: View {  
    var body: some View {  
        NavigationStack {  
            ZStack {  
                Color.green.opacity(0.25)  
                    .ignoresSafeArea()  
                Color.gray.opacity(0.25)  
            }  
            .navigationTitle("Background Color")  
        }  
    }  
}
```

The contents of the navigation stack is within the safe area.

Where the nav bar is on top is an "unsafe area".

If you want a color to go behind the navigation stack's title bar, then you will have to ignore that safe area edge (or all edges).

This gray color gives you a better idea where the safe area is and where the edge is against the navigation stack's title bar.

Custom Background



```
struct Navigation_CustomBackground: View {  
    var body: some View {  
        NavigationStack {  
            VStack {  
                Divider()  
                .background(  
                    LinearGradient(colors: [.green, .blue],  
                    startPoint: .leading,  
                    endPoint: .trailing)  
                    .opacity(0.5)  
                    .shadow(.drop(radius: 2, y: 2)),  
                    ignoresSafeAreaEdges: .top)  
                Spacer()  
            }  
            .navigationTitle("Custom Background")  
        }  
    }  
}
```

This background modifier will automatically ignore all safe area edges. That's why it expands into the nav bar area to create your nav background.

Use this optional parameter to specify which edges the background should ignore. By default it ignores all edges.



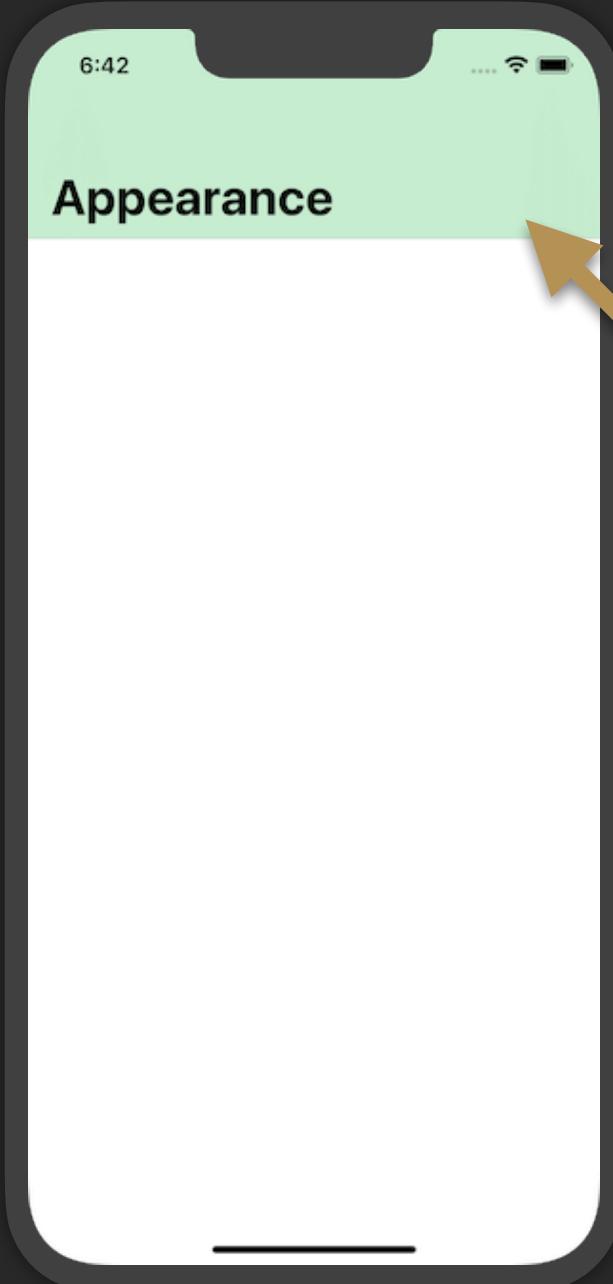
Note: This will change just this screen's navigation stack, not all nav stack though. To change all navigation stacks, see next page.



UINavigationBarAppearance

```
struct Navigation_UINavigationBarAppearance: View {  
    var body: some View {  
        NavigationStack {  
            VStack {  
                ...  
            }  
            .navigationTitle("Appearance")  
            .font(.title)  
        }  
        .onAppear {  
            let appearance = UINavigationBarAppearance()  
  
            appearance.backgroundColor = UIColor(Color.green.opacity(0.25))  
  
            UINavigationBar.appearance().standardAppearance = appearance  
            UINavigationBar.appearance().scrollEdgeAppearance = appearance  
        }  
    }  
}
```

Use UINavigationBarAppearance to apply style/color to ALL navigation bars.

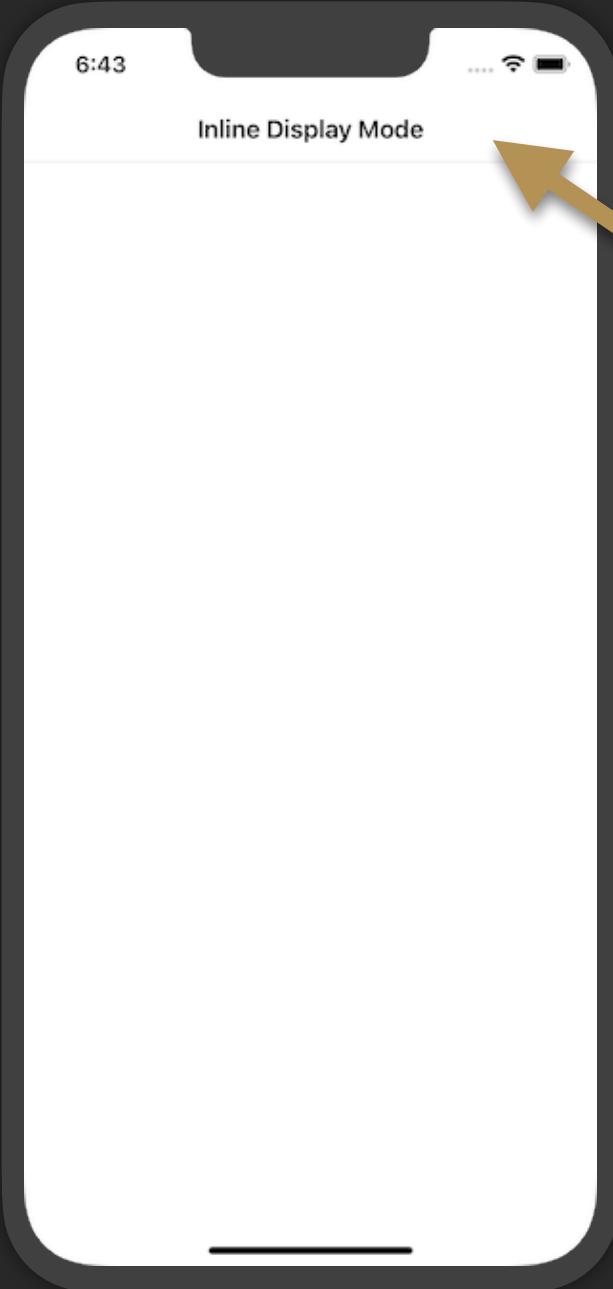


Note: You could also set this when the app starts on the very first view of your app.



Display Mode

```
struct Navigation_DisplayMode: View {  
    var body: some View {  
        NavigationStack {  
            VStack {  
                Divider()  
                Spacer()  
            }  
            .navigationTitle("Inline Display Mode")  
            .navigationBarTitleDisplayMode(.inline)  
        }  
    }  
}
```

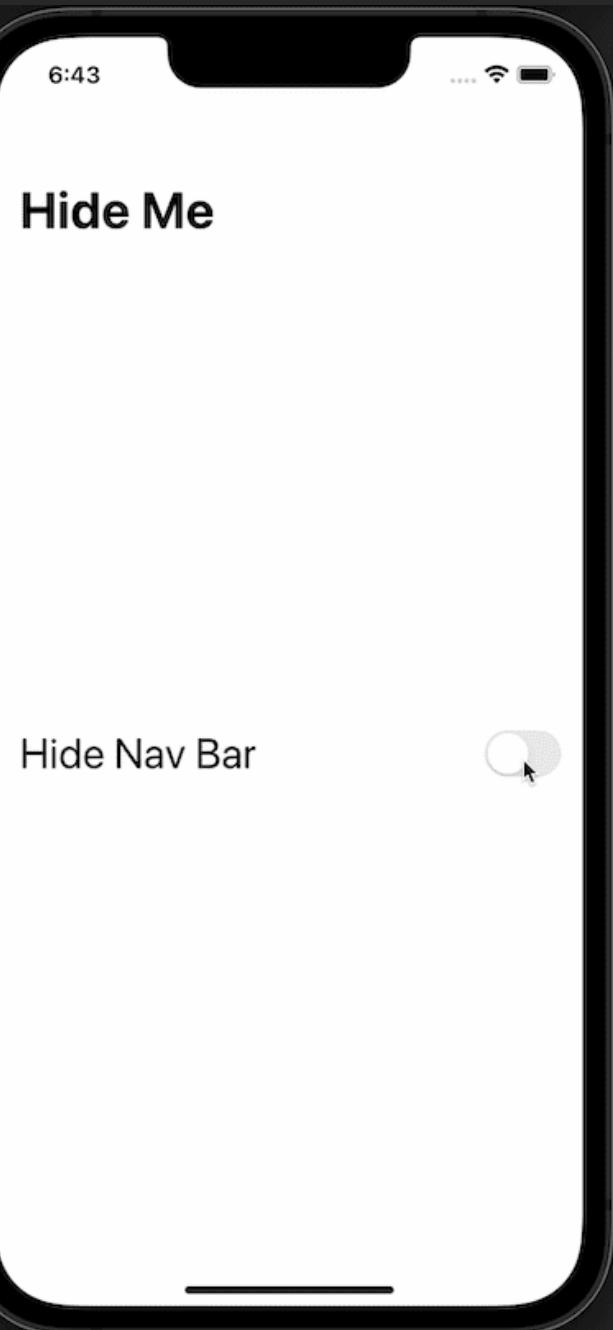


Using .inline will render the smaller nav bar.



Note: If you navigate from this screen, the next one will also have an inline bar title.

NavigationBarHidden



```
struct Navigation_BarHidden: View {  
    @State private var isHidden = false  
  
    var body: some View {  
        NavigationStack {  
            Toggle("Hide Nav Bar", isOn: $isHidden)  
                .font(.title)  
                .padding()  
                .navigationBarHidden(isHidden)  
                .navigationTitle("Hide Me")  
        }  
    }  
}
```

Use this modifier if you want to hide the navigation bar. This is helpful when you want to control navigation yourself.

Navigation Bar Items



Navigation Bar Buttons

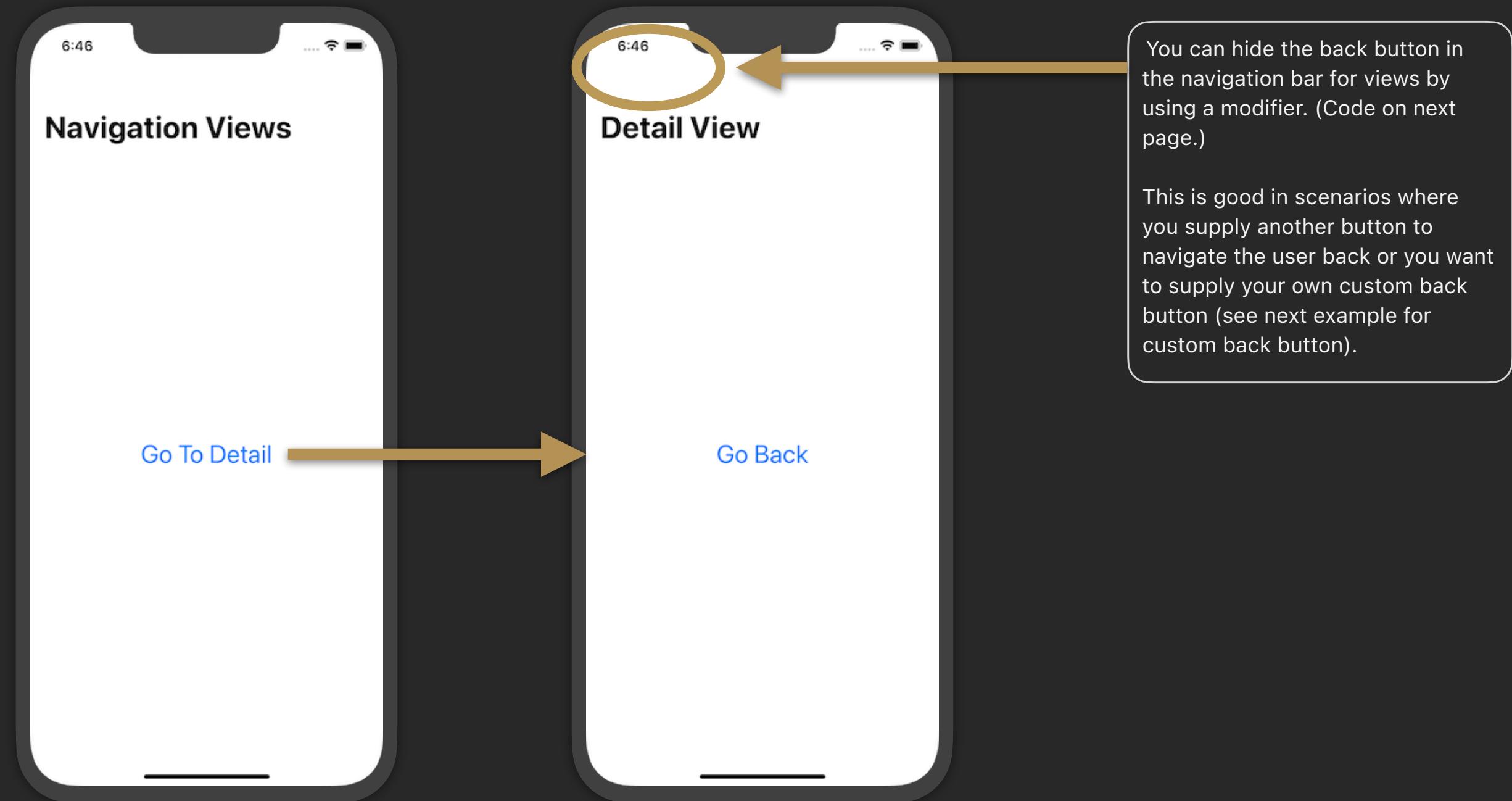
```
struct Navigation_NavBarItems : View {  
    var body: some View {  
        NavigationStack {  
            VStack {  
            }  
            .navigationTitle("Navigation Bar Buttons")  
            .toolbar {  
                ToolbarItem(placement: .navigationBarLeading) {  
                    Button(action: {}) {  
                        Image(systemName: "bell.fill")  
                            .padding(.horizontal)  
                    }  
                }  
                ToolbarItem(placement: .navigationBarTrailing) {  
                    Button("Actions", action: { })  
                }  
            }  
            .tint(.pink)  
        }  
    }  
}
```

You can add navigation bar buttons to the leading or trailing (or both) sides of a navigation bar.

For more ways on how to use the toolbar modifier, go to Controls Modifiers > Toolbar section.

Use tint to change the color of the buttons.

NavigationBarBackButtonHidden





(Code) NavigationBarBackButtonHidden

```
// First Screen
struct Navigation_BackButtonHidden: View {
    var body: some View {
        NavigationStack {
            NavigationLink("Go To Detail", destination: BackButtonHiddenDetail())
                .font(.title)
                .navigationTitle("Navigation Views")
        }
    }
}
```

```
// Second Screen
struct BackButtonHiddenDetail: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        Button("Go Back") {
            dismiss()
        }
        .font(.title)
        .navigationTitle("Detail View")
        // Hide the back button
        .navigationBarBackButtonHidden(true)
    }
}
```

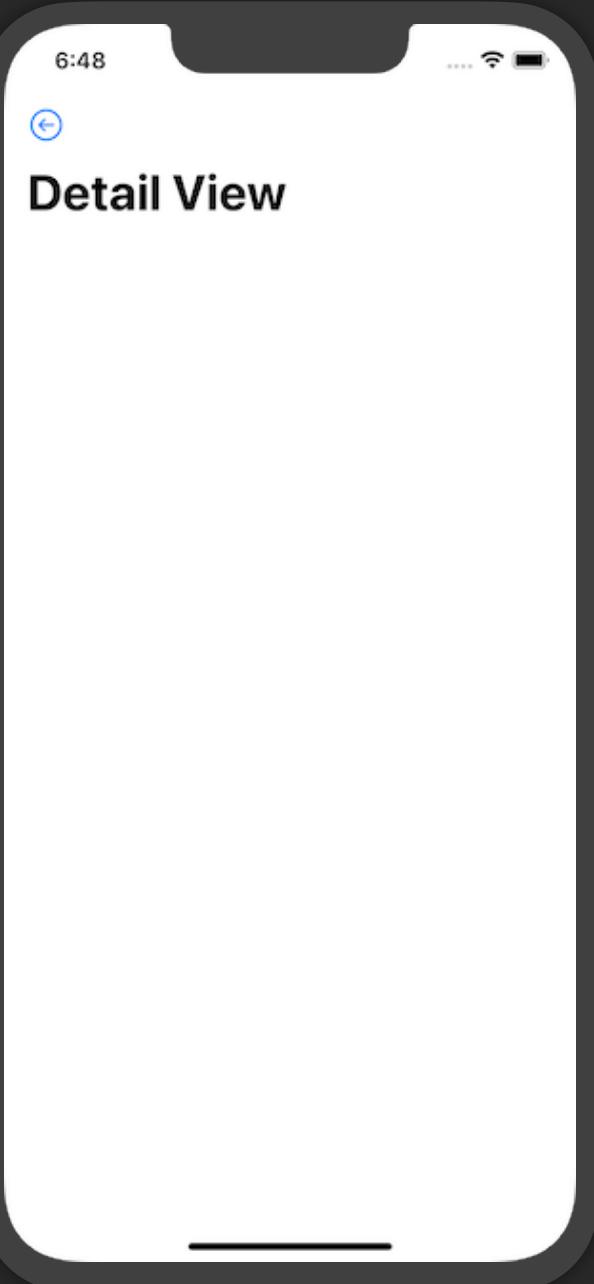
Use NavigationLink to navigate to a new screen.
More about NavigationLink in the next section.

This will allow you to navigate backward.

Dismissing what is being presented will navigate you back to the previous screen.

The back button is hidden. This allows you to create a custom back button where you might want to add logic to it.

Custom Back Button



```

struct Navigation_CustomBackButton: View {
    var body: some View {
        NavigationStack {
            NavigationLink("Go To Detail",
                           destination: Navigation_CustomBackButton_Detail())
                .font(.title)
                .navigationTitle("Navigation Views")
        }
    }
    // Second Screen
    struct Navigation_CustomBackButton_Detail: View {
        @Environment(\.dismiss) var dismiss

        var body: some View {
            VStack {
                .navigationTitle("Detail View")
                .navigationBarBackButtonHidden(true)
                .toolbar {
                    ToolbarItem(placement: .navigationBarLeading) {
                        Button(action: {
                            dismiss()
                        }) {
                            Label("Back", systemImage: "arrow.left.circle")
                        }
                    }
                }
            }
        }
    }
}

```

Hide the system back button and then use toolbar modifier to add a leading button.

Toolbar buttons can't show text and images at the same time.

If you want text and image, then hide the whole nav bar (previous example).



Warning: By hiding the back button, you will lose the ability to swipe back to the previous screen.



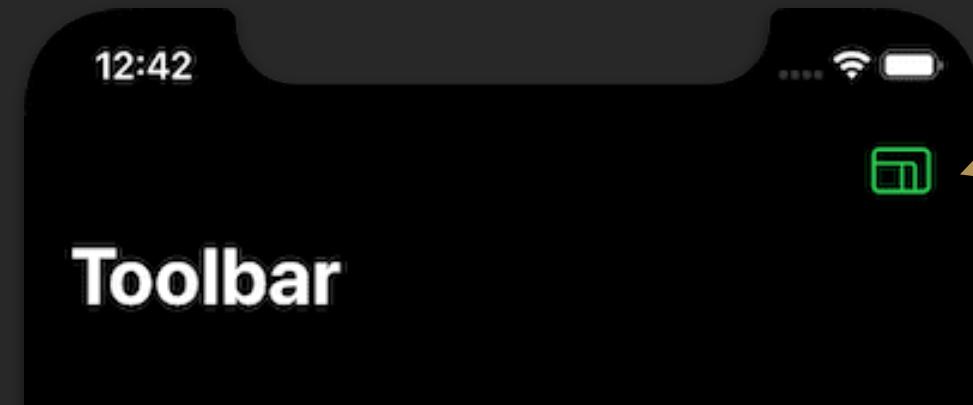
Customizing the Nav Bar

iOS 14

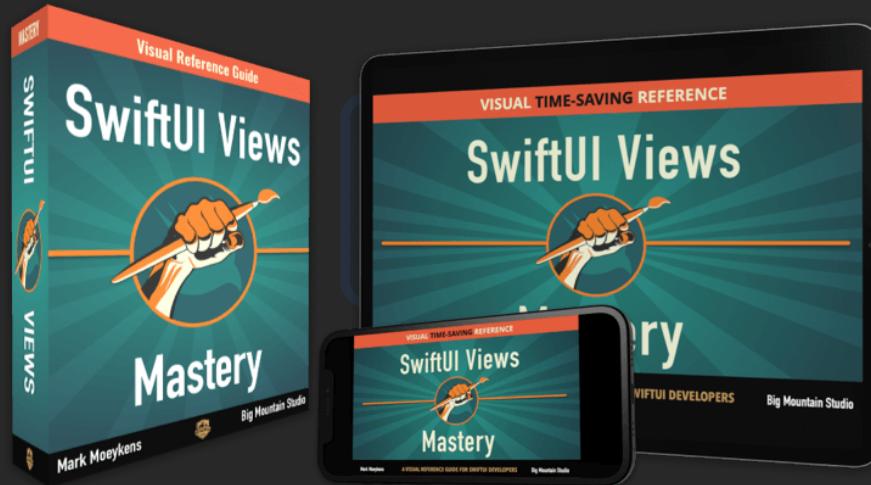


Learn about more options for adding buttons and customizing the NavigationView with the toolbar.

To explore this functionality, go to **Controls Modifier > Toolbar**.



NavigationSplitView



This SwiftUI chapter is locked in this
preview.

You have probably seen on an iPhone or macOS an app that has 2 or 3 vertical panes. In the far left pane you can tap on items that shows more details to the right. This is a common navigational pattern that can be achieved with the navigation split view.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

NavigationLink

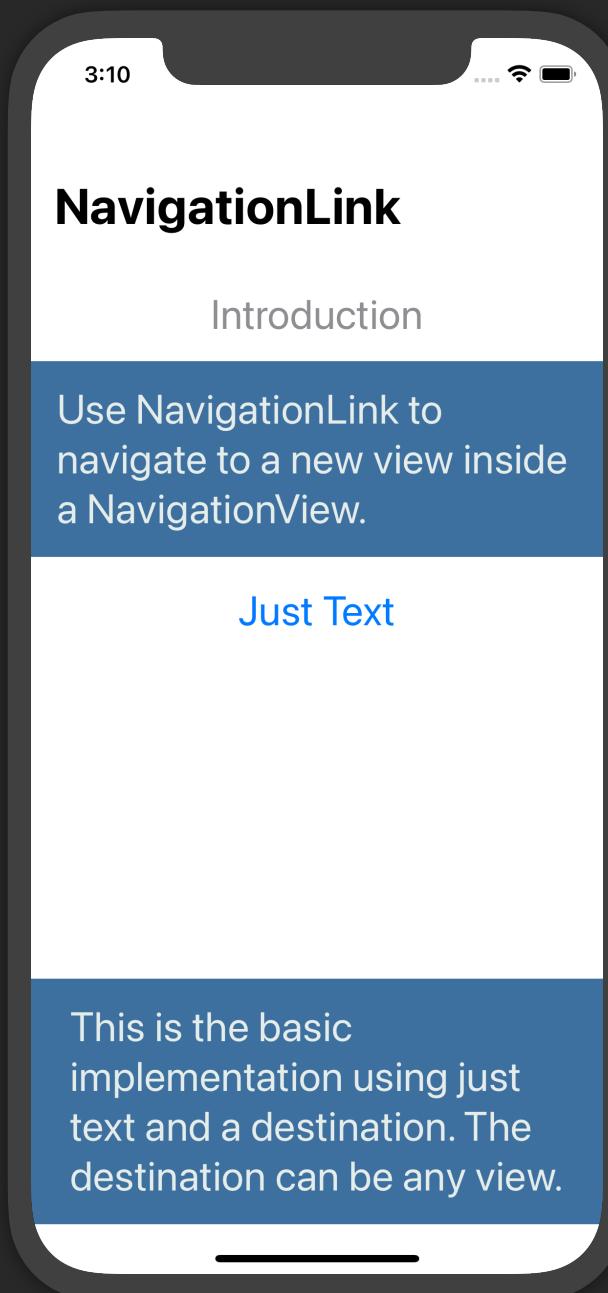


The NavigationLink is your way to navigate to another view. It ONLY works inside of a NavigationView. The appearance is just like a Button. You can customize it just like you can customize a Button too.

This is a pull-In view.



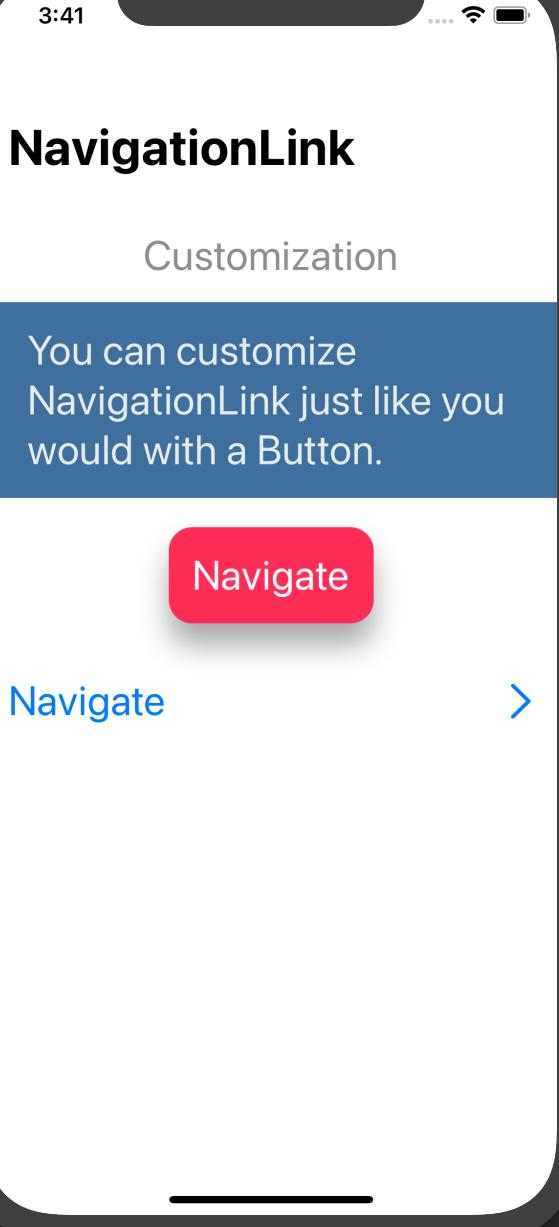
Introduction



```
struct NavLink_Intro: View {  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("",  
                           subtitle: "Introduction",  
                           desc: "Use NavigationLink to navigate to a new view inside a  
                           NavigationView.",  
                           back: Color("Theme3ForegroundColor"),  
                           textColor: Color("Theme3BackgroundColor"))  
                NavigationLink("Just Text", destination: SecondView())  
                Spacer()  
                DescView(desc: "This is the basic implementation using just text and a  
                           destination. The destination can be any view.",  
                           back: Color("Theme3ForegroundColor"),  
                           textColor: Color("Theme3BackgroundColor"))  
            }  
            .navigationTitle("NavigationLink")  
        }  
        .font(.title)  
    }  
  
    struct SecondView: View {  
        var body: some View {  
            VStack {  
                Text("View 2")  
                    .font(.largeTitle)  
                Spacer()  
            }  
        }  
    }  
}
```

Define text and a view for the destination.

Customization



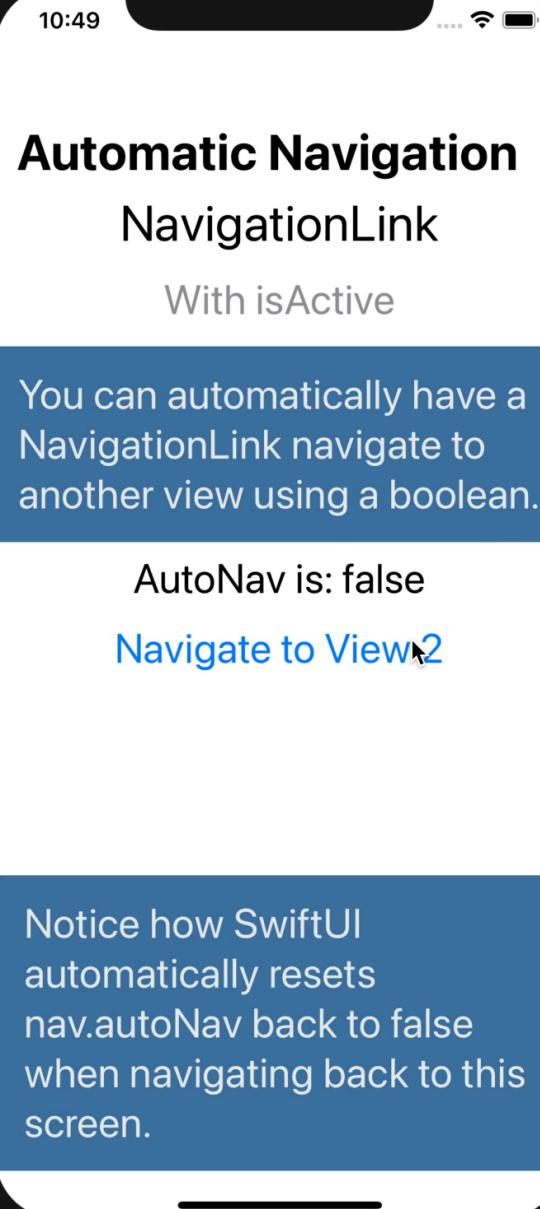
```
struct NavLink_Customization: View {  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("",  
                          subtitle: "Customization",  
                          desc: "You can customize NavigationLink just like you would with a  
                          Button.",  
                          back: Color("Theme3ForegroundColor"),  
                          textColor: Color("Theme3BackgroundColor"))  
  
                NavigationLink(destination: SecondView()) {  
                    Text("Navigate")  
                        .foregroundColor(.white)  
                        .padding()  
                        .background(RoundedRectangle(cornerRadius: 16)  
                                    .shadow(radius: 10, y: 15))  
                }  
                .accentColor(.pink)  
  
                NavigationLink(destination: SecondView()) {  
                    HStack {  
                        Text("Navigate")  
                        Spacer()  
                        Image(systemName: "chevron.right")  
                    }  
                    .padding()  
                }  
                Spacer()  
            }  
            .navigationTitle("NavLink")  
        }  
        .font(.title)  
    }  
}
```



The trailing closure is the label parameter. This allows you to compose any view that will navigate you.

Tip: Try to keep your views and modifiers within the closure. Like the Button, anything inside will fade when tapped.

With isActive



```
struct NavLink_WithIsActive: View {
    @State var nav = false

    var body: some View {
        NavigationView {
            VStack {
                HeaderView("NavigationView",
                           subtitle: "With isActive",
                           desc: "You can automatically have a NavigationLink navigate to another view using a boolean.",
                           back: Color("Theme3ForegroundColor"),
                           textColor: Color("Theme3BackgroundColor"))
                Text("AutoNav is: \(nav.description)")

                NavigationLink(destination: NavLink_View2(nav: $nav), isActive: $nav,
                               label: { EmptyView() })

                Button("Navigate to View 2") {
                    nav = true
                }
                Spacer()
                DescView(desc: "Notice how SwiftUI automatically resets nav.autoNav back to false when navigating back to this screen.",
                         back: Color("Theme3ForegroundColor"),
                         textColor: Color("Theme3BackgroundColor"))
            }
            .font(.title)
            .navigationTitle("Automatic Navigation")
        }
    }
}
```

The NavigationLink here has no visual representation so you won't see it in the UI.

NavigationLink(destination: NavLink_View2(nav: \$nav), isActive: \$nav, label: { EmptyView() })

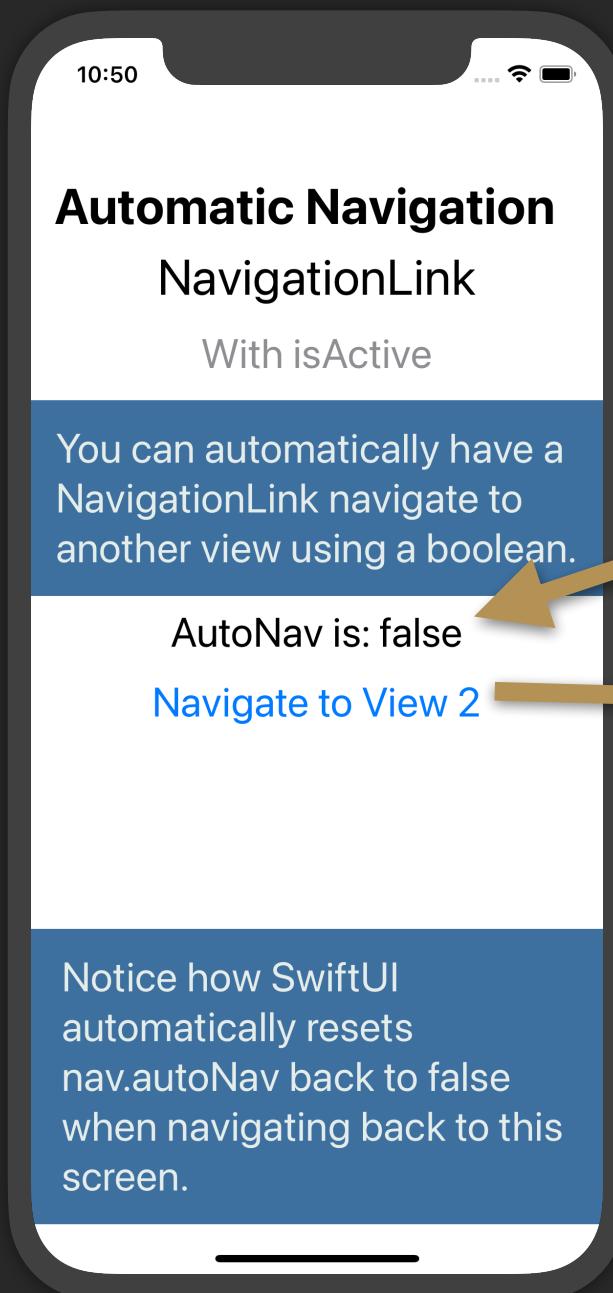
Button("Navigate to View 2") { nav = true }

Spacer()

DescView(desc: "Notice how SwiftUI automatically resets nav.autoNav back to false when navigating back to this screen.", back: Color("Theme3ForegroundColor"), textColor: Color("Theme3BackgroundColor"))

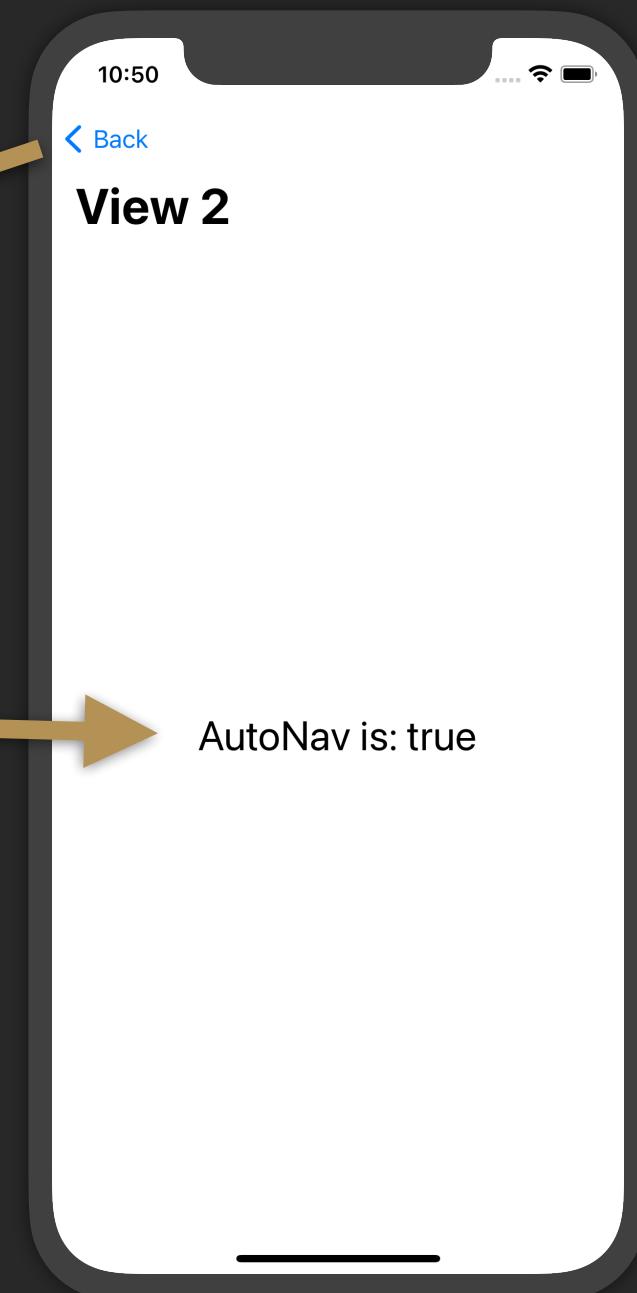
It's important to note that SwiftUI changes what you have bound to **isActive** back to **false** when you navigate back. See next page for more.

SwiftUI Resets isActive



When the Back button is tapped, SwiftUI changes what you have bound to `isActive` to `false` for you. You don't have to do anything.

When this button is tapped the property that is bound to `isActive` is set to `true` and navigation takes place. That bound property remains true until the Back button is tapped.

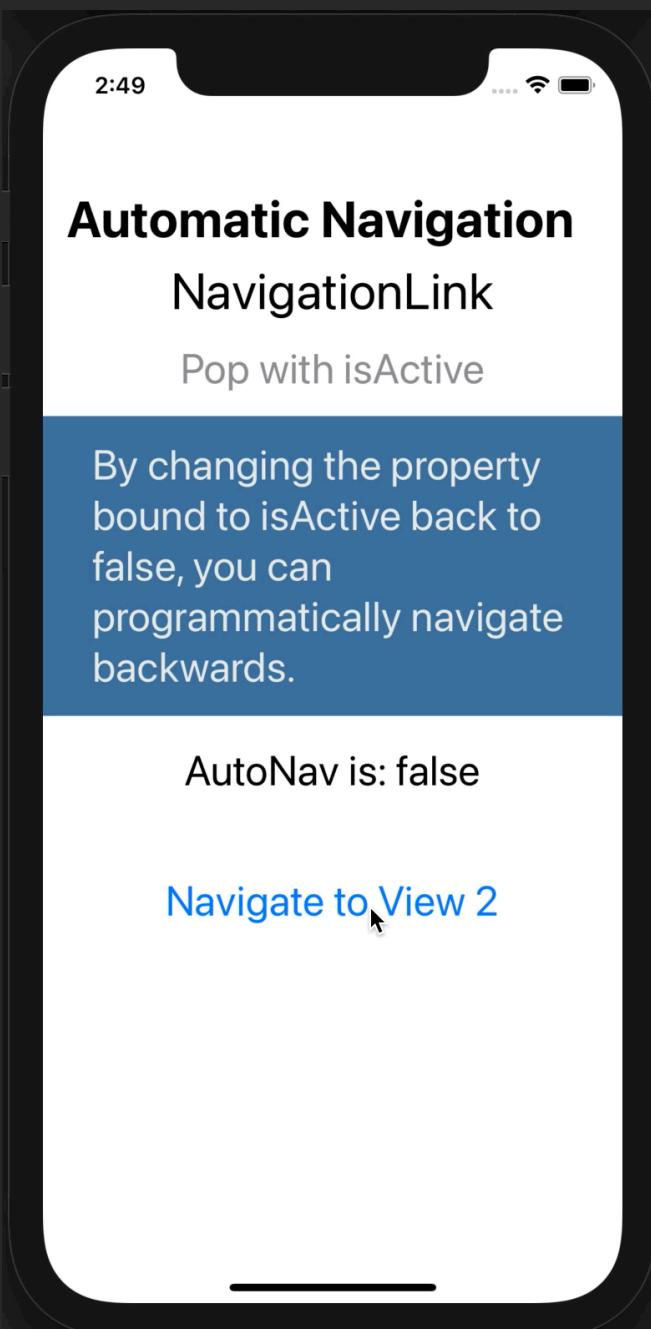




Using isActive to Pop Views

```
struct NavLink_WithIsActivePop: View {  
    @State var nav = false  
  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("NavigationView",  
                    subtitle: "Pop with isActive",  
                    desc: "By changing the property bound to isActive back to false, you  
can programmatically navigate backwards.",  
                    back: Color("Theme3ForegroundColor"),  
                    textColor: Color("Theme3BackgroundColor"))  
                Text("AutoNav is: \(nav.description)")  
  
                NavigationLink(destination: NavLink_View2(nav: $nav), isActive: $nav,  
                    label: { EmptyView() })  
  
                Button("Navigate to View 2") {  
                    nav = true  
                }  
                Spacer()  
            }  
            .font(.title)  
            .navigationTitle("Automatic Navigation")  
        }  
    }  
}
```

"Popping" a view just means removing it from the top so you navigate back to the previous view. You can also "pop" directly to the root view.



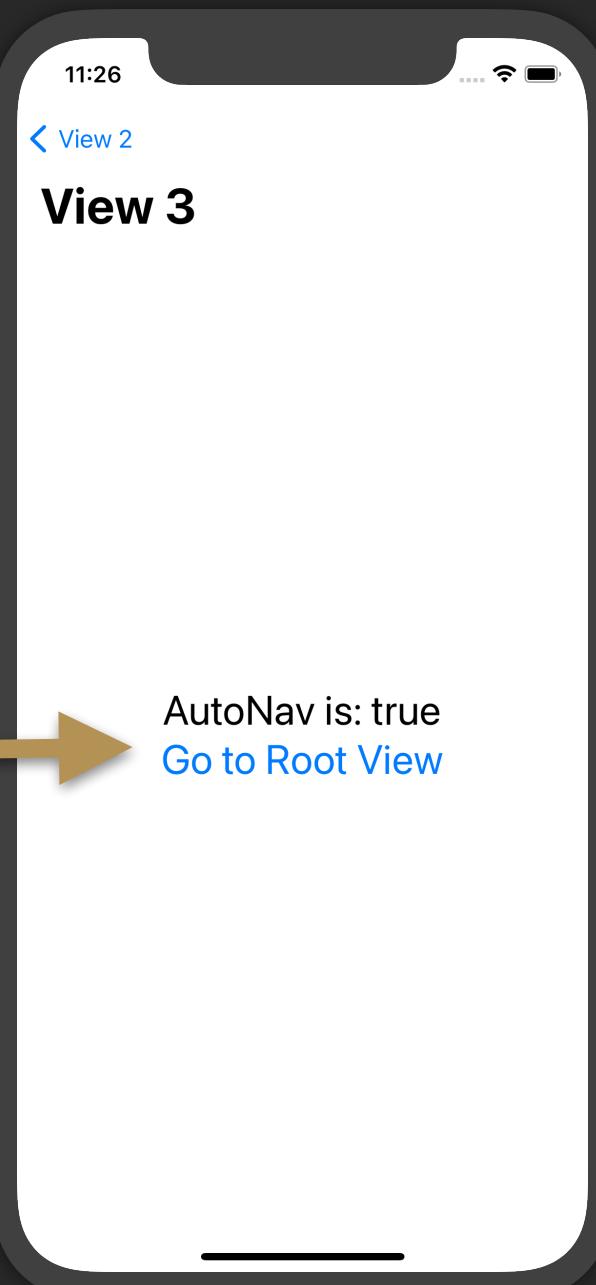
Using isActive to Pop Views

```
struct NavLink_SecondView: View {  
    @Binding var nav: Bool  
  
    var body: some View {  
        VStack {  
            Text("AutoNav is: \(nav.description)")  
            NavigationLink("Navigate to View 3", destination: NavLink_ThirdView(nav: $nav))  
        }  
        .font(.title)  
        .navigationTitle("View 2")  
    }  
}  
  
struct NavLink_ThirdView: View {  
    @Binding var nav: Bool  
  
    var body: some View {  
        VStack {  
            Text("AutoNav is: \(nav.description)")  
            Button("Go to Root View") {  
                nav.toggle()  
            }  
        }  
        .font(.title)  
    }  
}
```

Notice the second NavigationLink doesn't need to use the isActive parameter. But you do need to pass the original property bound to isActive.

By setting the same bound variable that was connected to isActive back to **false**, you can pop back to the root.

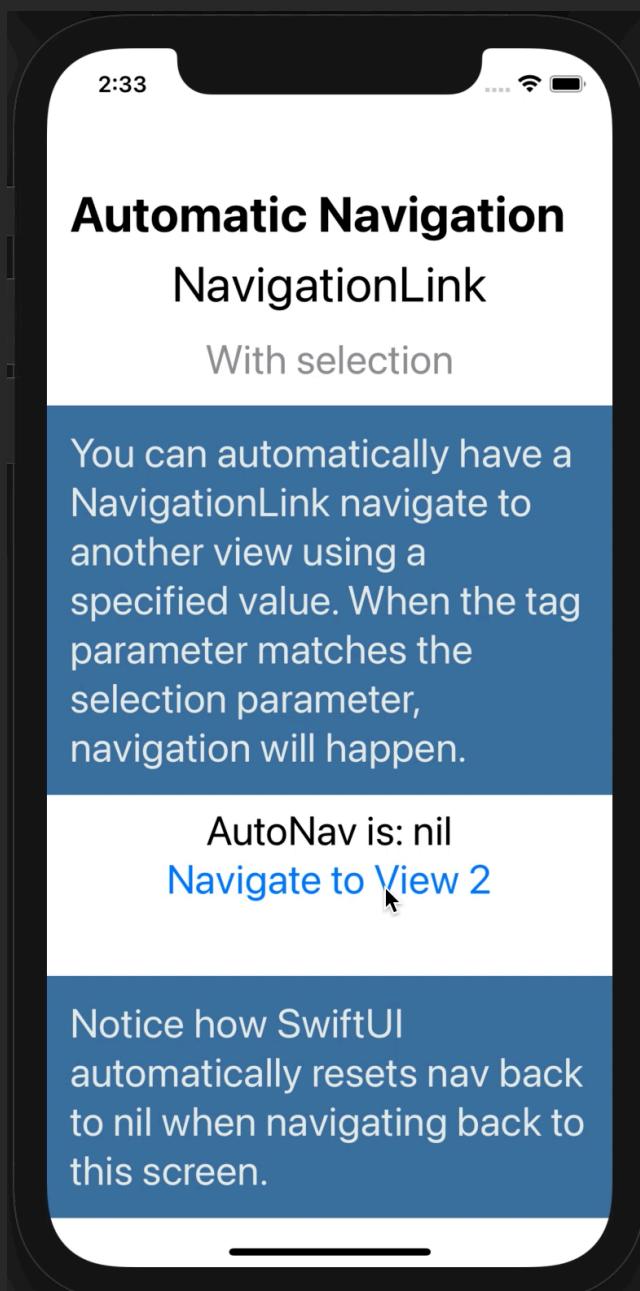
Note: In these examples I'm using **@State** and **@Binding**. You would more likely be using this with a view model.
To learn more, see the end of this book for information on how you can get the book "Working with Data in SwiftUI".



With selection

```
struct NavLink_WithSelection: View {  
    @State var nav: String?  
  
    var body: some View {  
        NavigationView {  
            VStack {  
                HeaderView("NavigationLink",  
                    subtitle: "With selection",  
                    desc: "You can automatically have a NavigationLink navigate to  
another view using a specified value. When the tag parameter matches the selection parameter,  
navigation will happen.",  
                    back: Color("Theme3ForegroundColor"),  
                    textColor: Color("Theme3BackgroundColor"))  
  
                Text("AutoNav is: \(nav ?? "nil")")  
  
                NavigationLink(destination: NavigationWithSelection(nav: $nav),  
                    tag: "To View 2", selection: $nav, label: {})  
  
                Button("Navigate to View 2") {  
                    nav = "To View 2"  
                }  
                Spacer()  
                DescView(desc: "Notice how SwiftUI automatically resets nav back to nil when  
navigating back to this screen.",  
                    back: Color("Theme3ForegroundColor"),  
                    textColor: Color("Theme3BackgroundColor"))  
            }  
            .font(.title)  
            .navigationTitle("Automatic Navigation")  
        }  
    }  
}
```

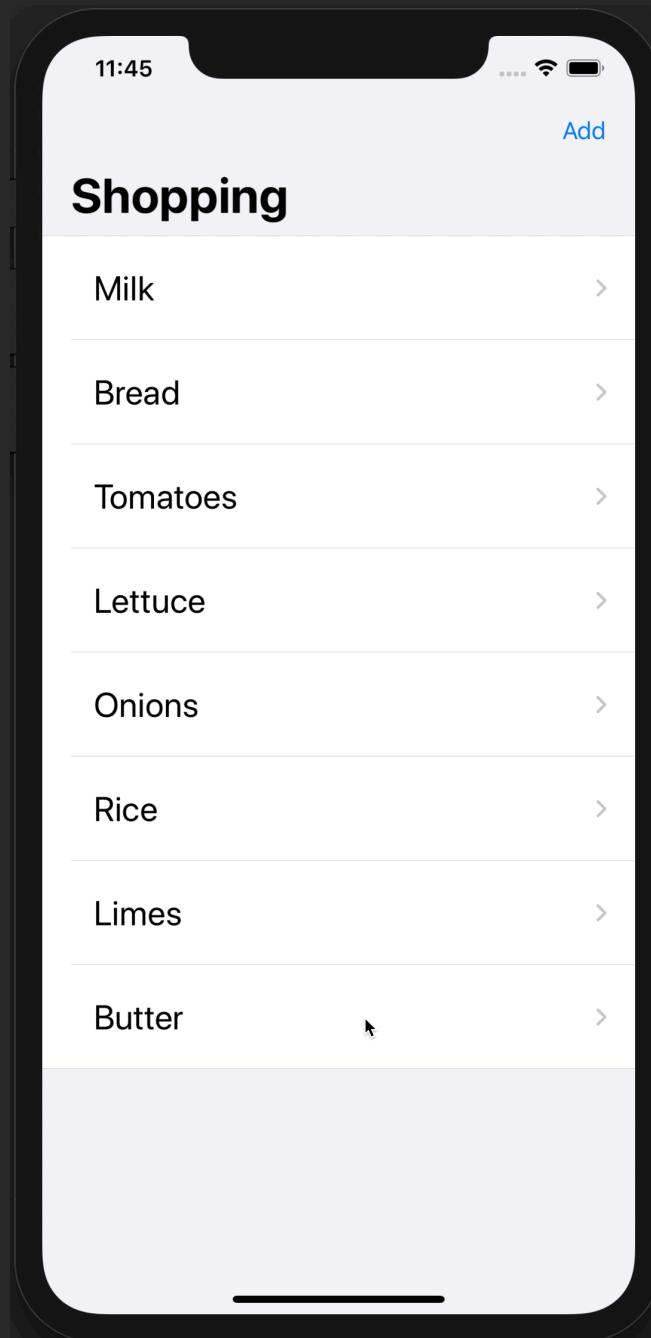
Basically, when the selection parameter value matches the value of the tag parameter, navigation happens.



It's important to note that SwiftUI changes what you have bound to **selection** back to **nil** when you



List In NavigationView with NavigationLink



```
struct Navigation_WithList: View {
    @State var data = ["Milk", "Bread", "Tomatoes", "Lettuce", "Onions", "Rice", "Limes"]

    var body: some View {
        NavigationView {
            List(data, id: \.self) { datum in
                NavigationLink(destination: ShoppingDetail(shoppingItem: datum)) {
                    Text(datum).font(Font.system(size: 24)).padding()
                }
            }
            .listStyle(.grouped)
            .navigationTitle("Shopping")
            .toolbar {
                ToolbarItem {
                    Button("Add", action: { data.append("New Shopping Item") })
                }
            }
        }
    }
}

struct ShoppingDetail: View {
    var shoppingItem: String!
    var body: some View {
        VStack {
            Text("Shopping List Details").font(.title)
            .frame(maxWidth: .infinity).padding()
            .background(Color("Theme3ForegroundColor"))
            .foregroundColor(Color("Theme3BackgroundColor"))
            Spacer()
            Text(shoppingItem).font(.title)
            Spacer()
        }.navigationTitle(shoppingItem)
    }
}
```

Navigate to detail view

Learn more about the toolbar and ToolbarItem in the "Controls Modifiers" chapter.



List Navigation with No NavigationView

```
struct Navigation_WithListNoNavBar: View {
    @State var books = ["The Way of Kings", "Words of Radiance", "Oathbringer"]

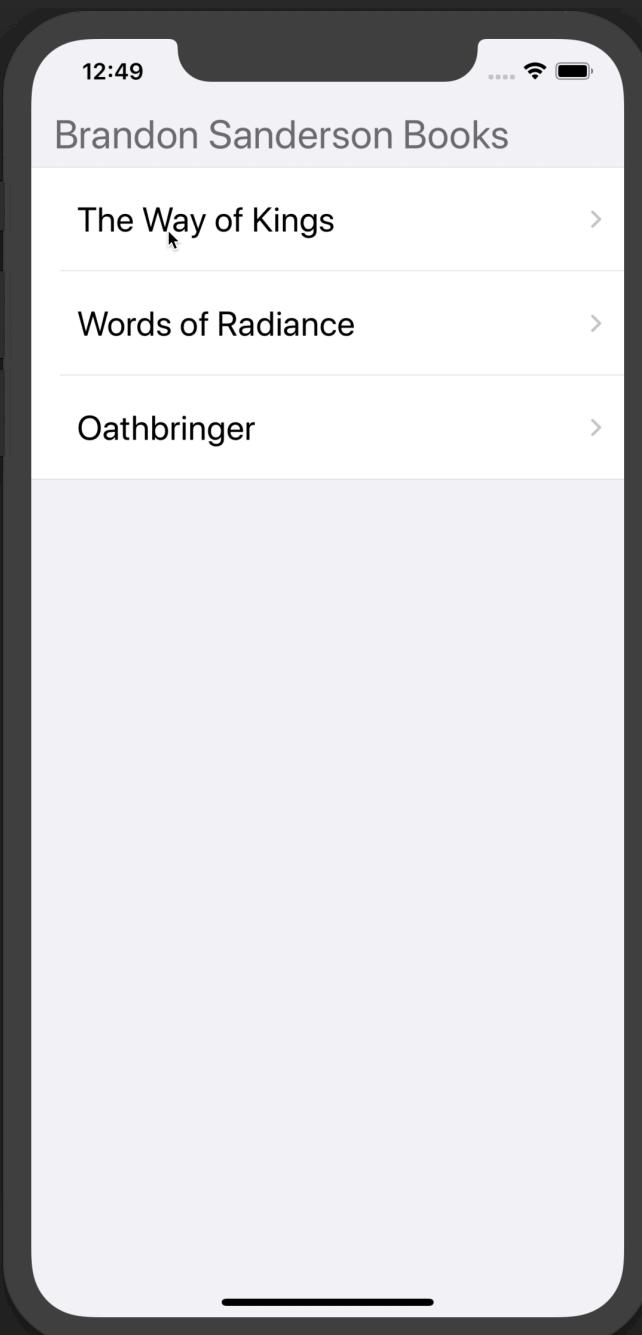
    var body: some View {
        NavigationView {
            List {
                Section(header: Text("Brandon Sanderson Books").font(.title)) {
                    ForEach(books, id: \.self) { book in
                        NavigationLink(destination: BookDetail(bookItem: book)) {
                            Text(book).font(Font.system(size: 24)).padding()
                        }
                    }
                }
                .listStyle(.grouped)
                .navigationTitle("Book List") // Your back button text
                .navigationBarHidden(true)
            }
        }
    }
}

struct BookDetail: View {
    var bookItem: String!

    var body: some View {
        VStack {
            Text("Book Details").font(.title)
                .frame(maxWidth: .infinity).padding()
                .background(Color("Theme3ForegroundColor"))
                .foregroundColor(Color("Theme3BackgroundColor"))
            Spacer()
            Text(bookItem).font(.title)
            Spacer()
        }.navigationTitle(bookItem)
    }
}
```

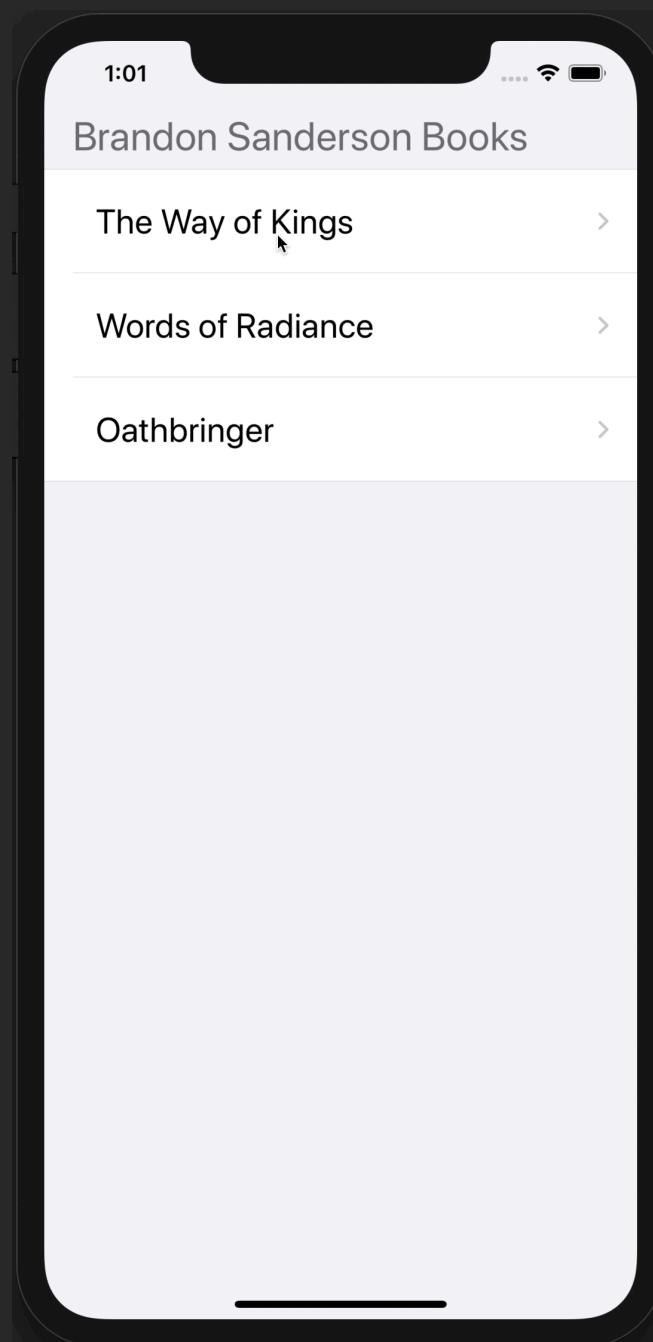
The NavigationLink is the whole row. It automatically adds the gray chevron icon.

Now this just looks like a regular list view but has navigation on the rows.





List Navigation with No NavigationView & No Back Button



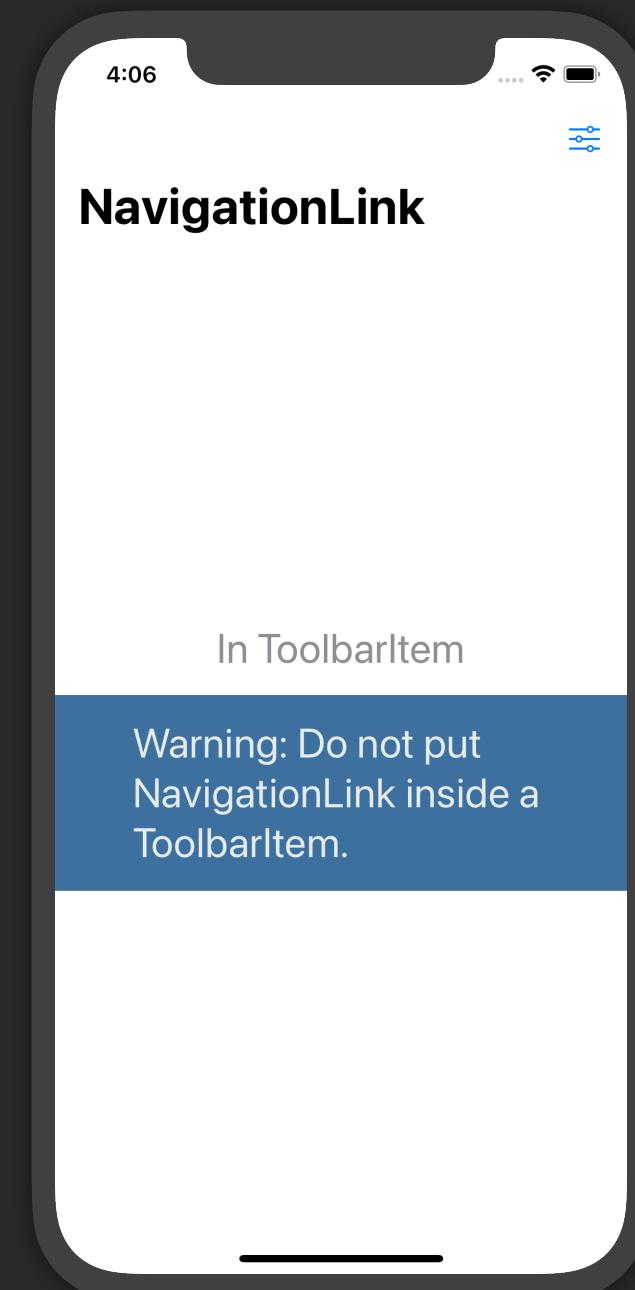
The first view will be exactly the same as the previous page. The detail view will be different though:

```
struct BookDetail_NoBack: View {  
    @Environment(\.presentationMode) var presentationMode  
    var bookItem: String!  
  
    var body: some View {  
        VStack {  
            Text("Book Details").font(.title)  
            .frame(maxWidth: .infinity).padding()  
            .background(Color("Theme3ForegroundColor"))  
            .foregroundColor(Color("Theme3BackgroundColor"))  
            Spacer()  
            Text(bookItem).font(.title)  
            Spacer()  
            Button(action: {  
                presentationMode.wrappedValue.dismiss()  
            }) {  
                HStack {  
                    Image(systemName: "arrow.left.circle")  
                    Text("Go Back")  
                }  
            }  
            Spacer()  
        }  
        .navigationTitle(bookItem)  
        .navigationBarHidden(true)  
    }  
}
```

This will allow you to navigate backward.



Warning: NavigationLink in ToolbarItem



```
struct NavLink_InToolbarItem: View {  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("",  
                           subtitle: "In ToolbarItem",  
                           desc: "Warning: Do not put NavigationLink inside a ToolbarItem.",  
                           back: Color("Theme3ForegroundColor"),  
                           textColor: Color("Theme3BackgroundColor"))  
            }  
            .toolbar {  
                ToolbarItem {  
                    // Do not do this  
                    NavigationLink(destination: Text("Settings"),  
                                   label: {  
                                       Image(systemName: "slider.horizontal.3")  
                                   })  
                }  
            }.navigationTitle("NavLink")  
            .font(.title)  
        }  
    }  
}
```

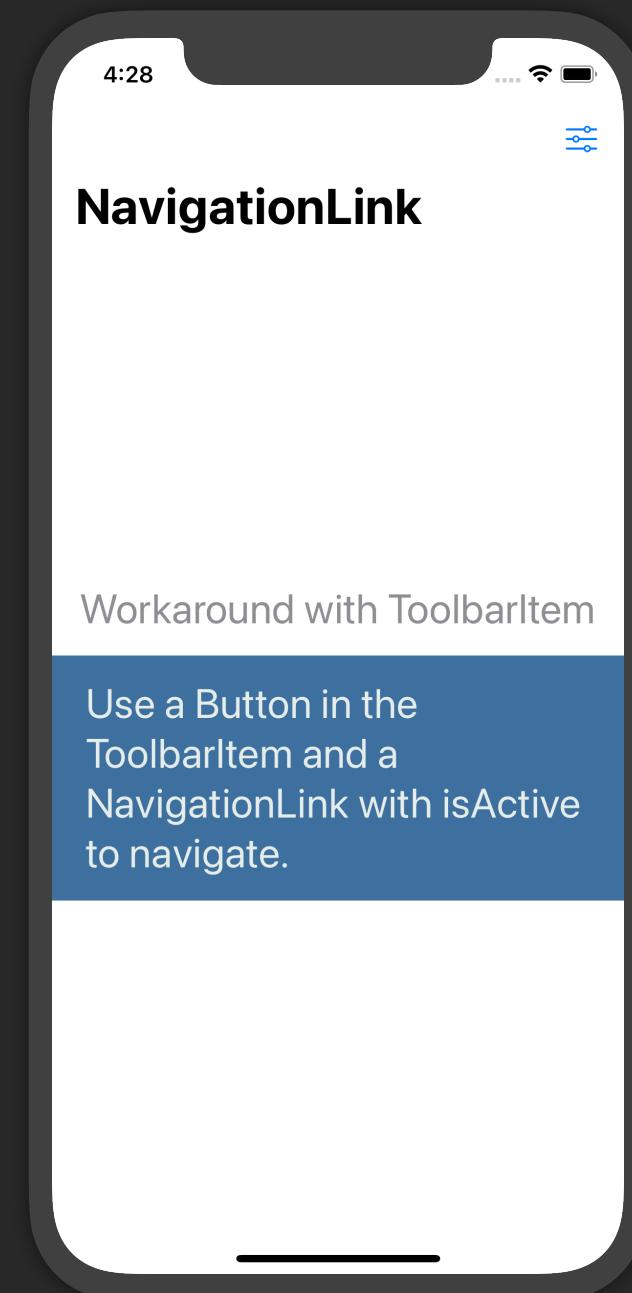


Without getting too technical, the NavigationLink isn't the type of view that SwiftUI knows when and when not to recreate. So it is recreated all the time. Xcode will not warn you against this. And, whether intentional or a bug, **it can navigate multiple times or not at all depending on the iOS version.** It might look like it works in Preview, but be sure to test on Simulator and on a device.

Workaround: There is another way around this though which you will see on the next page.



Workaround: NavigationLink in ToolbarItem



```
struct NavLink_InToolbarWorkaround: View {  
    @State private var navigate = false  
  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("",  
                          subtitle: "Workaround with ToolbarItem",  
                          desc: "Use a Button in the ToolbarItem and a NavigationLink with  
                                 isActive to navigate.",  
                          back: Color("Theme3ForegroundColor"),  
                          textColor: Color("Theme3BackgroundColor"))  
                NavigationLink(destination: Text("Settings"),  
                               isActive: $navigate,  
                               label: {})  
            }  
            .toolbar {  
                ToolbarItem {  
                    Button(action: { navigate = true }) {  
                        Image(systemName: "slider.horizontal.3")  
                    }  
                }  
            }  
            .navigationTitle("NavigationView")  
            .font(.title)  
        }  
    }  
}
```

You can use a NavigationLink with no visual element to do the navigation for you.

Using a Button in a ToolbarItem is the preferred way to do this.



Using isDetailLink with iPads

```
struct NavLink_IsDetailLink: View {
    var body: some View {
        NavigationView {
            VStack(spacing: 20) {
                HeaderView("", subtitle: "isDetailLink",
                           desc: "By default, when you navigate on an iPad, your first view will
                           be on the left and your new view will be on the right. The view on the right is called the
                           'Detail'. You can change this behavior by using the isDetailLink modifier.",
                           back: Color("Theme3ForegroundColor"),
                           textColor: Color("Theme3BackgroundColor"))
                NavigationLink(
                    destination: DetailLinkView(),
                    label: {
                        Text("Show Detail Inside")
                    })
                    .isDetailLink(false)
            }
            .navigationTitle("NavLink")
            .font(.title)
        }
    }
}

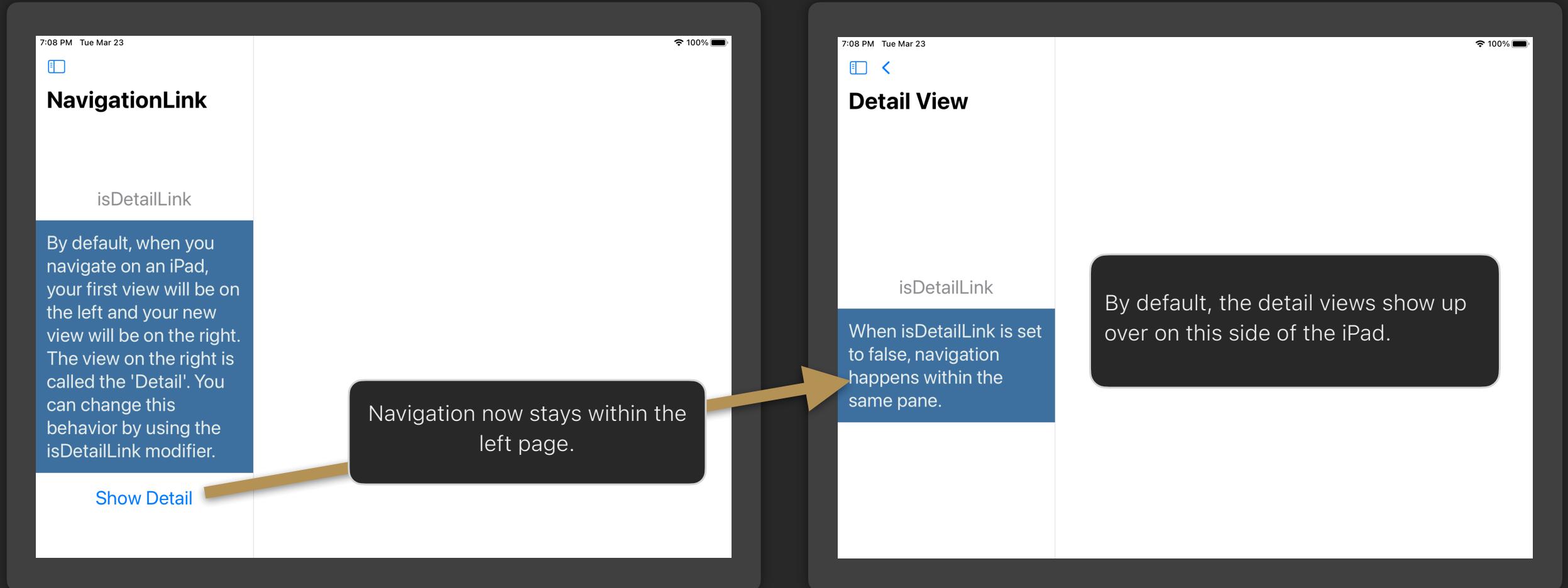
struct DetailLinkView: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("", subtitle: "isDetailLink",
                           desc: "When isDetailLink is set to false, navigation happens within the
                           same pane.",
                           back: Color("Theme3ForegroundColor"),
                           textColor: Color("Theme3BackgroundColor"))
        }
        .navigationTitle("Detail View")
        .font(.title)
    }
}
```

Let SwiftUI know you don't want to open the view on the right pane when on iPad.

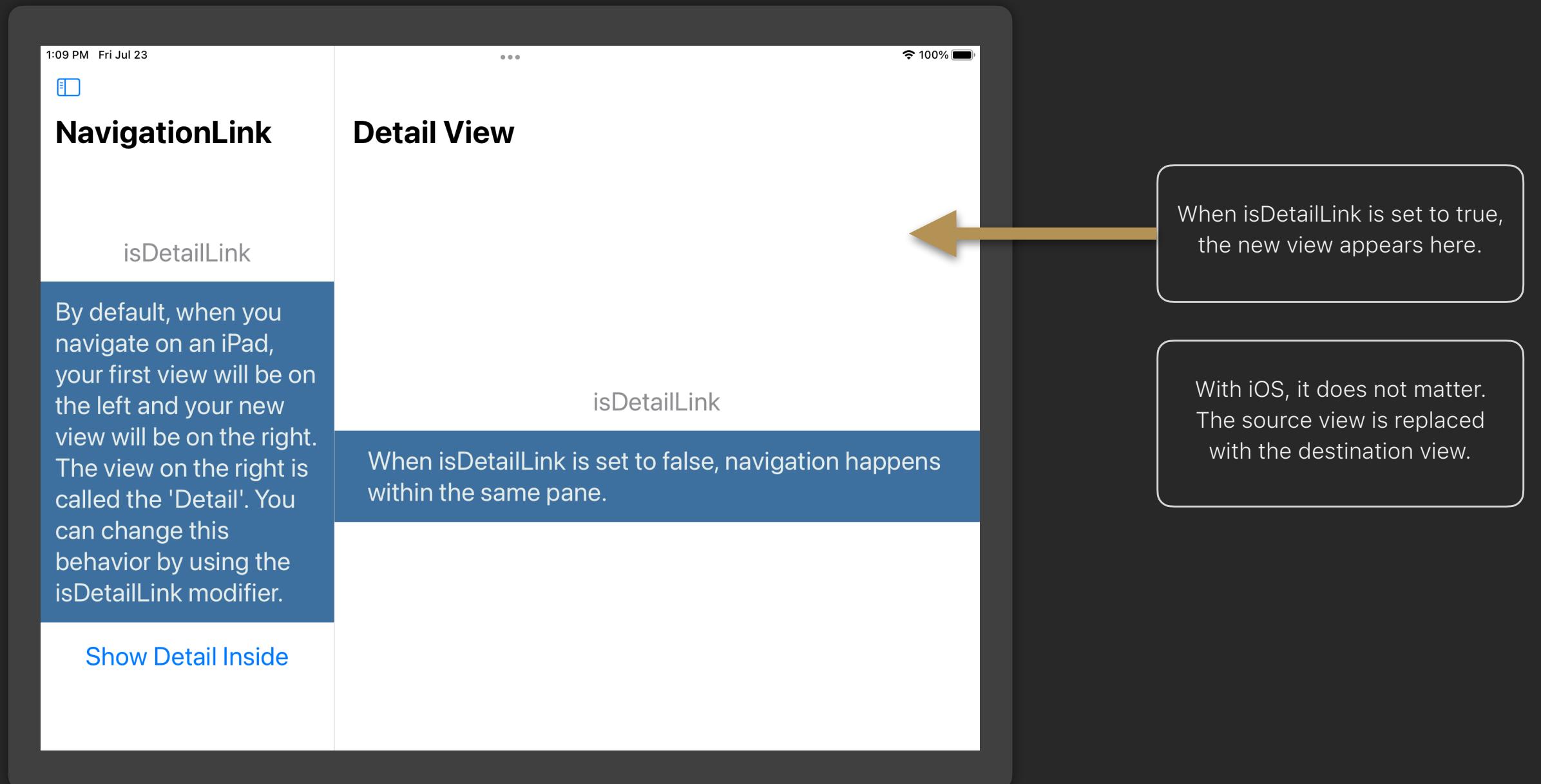
See result on next page...



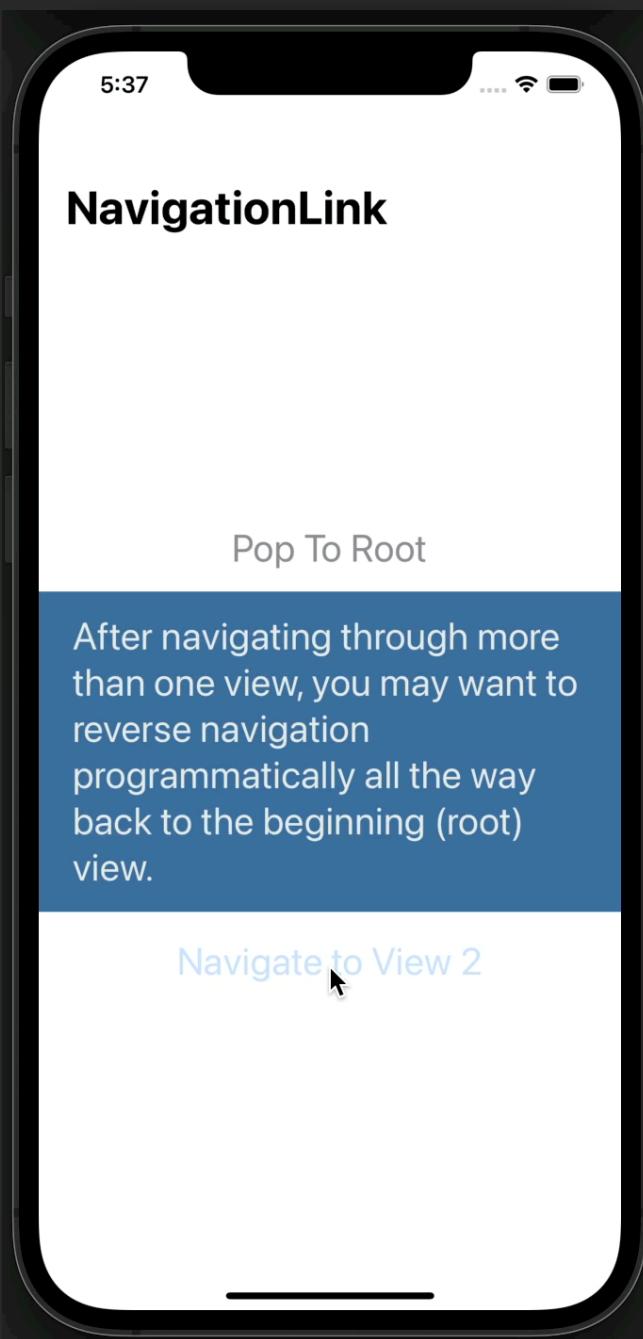
When isDetailLink is false



When isDetailLink is true



Pop to Root



```
struct NavLink_PopToRoot: View {  
    @State private var isActive = false  
  
    var body: some View {  
        NavigationView {  
            VStack(spacing: 20) {  
                HeaderView("",  
                          subtitle: "Pop To Root",  
                          desc: "After navigating through more than one view, you may want to  
reverse navigation programmatically all the way back to the  
beginning (root) view.",  
                          back: Color("Theme3ForegroundColor"),  
                          textColor: Color("Theme3BackgroundColor"))  
  
                NavigationLink(  
                    destination: NavLinkView2(isActive: $isActive),  
                    isActive: $isActive,  
                    label: {  
                        Text("Navigate to View 2")  
                    })  
            }  
            .navigationTitle("NavigationLink")  
            .font(.title)  
        }  
    }  
}
```

The key is to use this isActive parameter and then pass that value to the views that should pop back to the root.

Pop to Root (The other views)

```
struct NavLinkView2: View {  
    @Binding var isActive: Bool  
  
    var body: some View {  
        VStack {  
            NavigationLink("Navigate to View 3", destination: NavLinkView3(isActive: $isActive))  
        }  
        .navigationTitle("View 2")  
        .font(.title)  
    }  
}  
  
struct NavLinkView3: View {  
    @Binding var isActive: Bool  
  
    var body: some View {  
        VStack {  
            Button(action: {  
                isActive.toggle()  
            }, label: {  
                Text("Go Back to Root View")  
            })  
        }  
        .navigationTitle("View 3")  
        .font(.title)  
    }  
}
```

Pass in the value for isActive but only use it to pop back to root. Don't use it to open another NavigationLink.

This is where the magic happens. Set the original isActive binding to false and it will pop all the way back to the root.

ios 14

OutlineGroup



This SwiftUI chapter is locked in this

OutlineGroups gives you an easy way to present hierarchical data. It is very similar to using a List with the children parameter. Except this container view does not scroll. It's probably best for limited data.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Picker



To get or set a value for the Picker, you need to bind it to a variable. This variable is then passed into the Picker's initializer. Then, all you need to do is change this bound variable's value to select the row you want to show in the Picker. Or read the bound variable's value to see which row is currently selected. One thing to note is that this variable is actually bound to the Picker row's **tag property** which you will see in the following pages.

Introduction



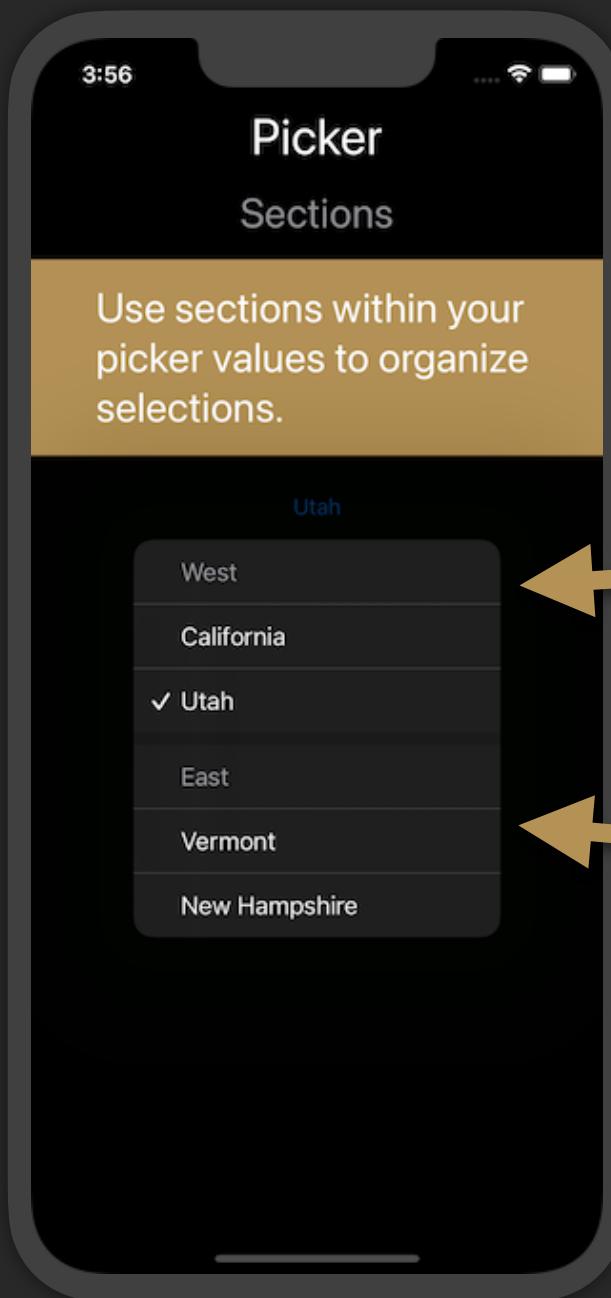
```
struct Picker_Intro : View {  
    @State private var favoriteState = 1  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Picker",  
                      subtitle: "Introduction",  
                      desc: "You can associate a property with the picker rows' tag values.")  
  
            Picker("States", selection: $favoriteState) {  
                Text("California").tag(0)  
                Text("Utah").tag(1)  
                Text("Vermont").tag(2)  
            }  
            Spacer()  
        .font(.title)  
    }  
}
```

Starting in iOS 15, the default picker style will look like the menu style.
(Before, the default was the wheel style.)

iOS 15

This picker is actually binding the tag values to the **favoriteState** property.

Sections



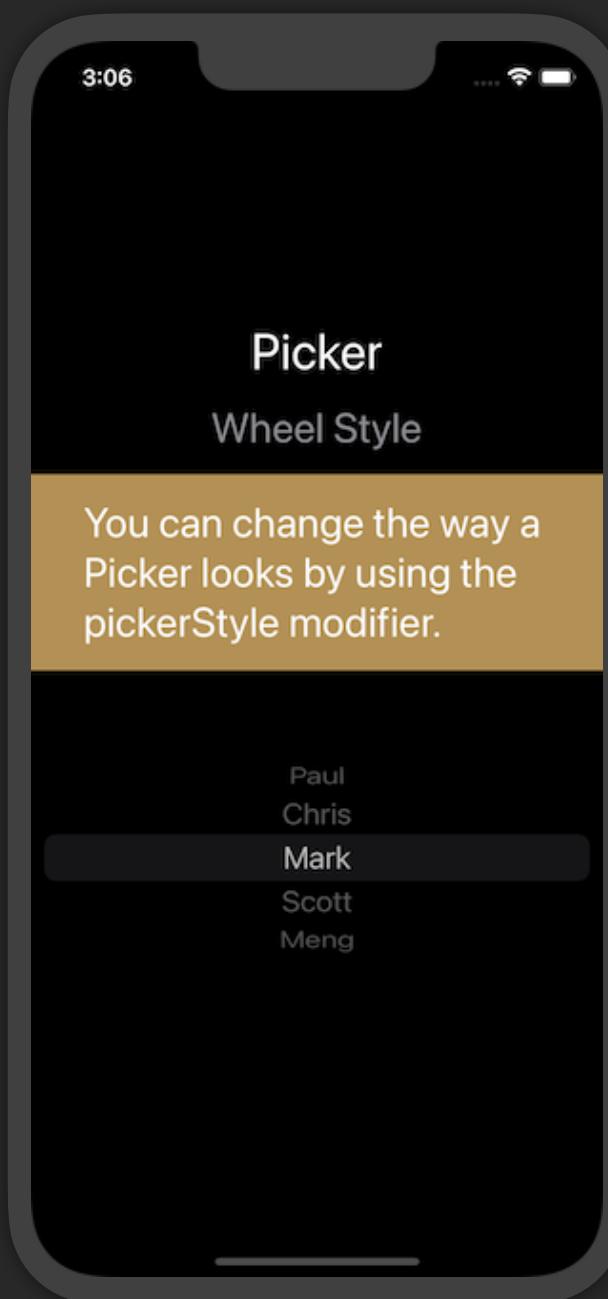
```
struct Picker_Sections: View {  
    @State private var favoriteState = 1  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Picker",  
                      subtitle: "Sections",  
                      desc: "Use sections within your picker values to organize selections.")  
  
            Picker("States", selection: $favoriteState) {  
                Section {  
                    Text("California").tag(0)  
                    Text("Utah").tag(1)  
                } header: {  
                    Text("West")  
                }  
  
                Section {  
                    Text("Vermont").tag(2)  
                    Text("New Hampshire").tag(3)  
                } header: {  
                    Text("East")  
                }  
            }  
            Spacer()  
        }  
        .font(.title)  
    }  
}
```

Wheel Style

```
struct Picker_Wheel: View {  
    @State private var yourName = "Mark"  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Picker",  
                      subtitle: "Wheel Style",  
                      desc: "You can change the way a Picker looks by using the pickerStyle modifier.")  
  
            Picker("Name", selection: $yourName) {  
                Text("Paul").tag("Paul")  
                Text("Chris").tag("Chris")  
                Text("Mark").tag("Mark")  
                Text("Scott").tag("Scott")  
                Text("Meng").tag("Meng")  
            }  
            .pickerStyle(.wheel)  
        }  
        .font(.title)  
    }  
}
```

This will be the default value selected in the Picker.

Set the style right on the Picker.



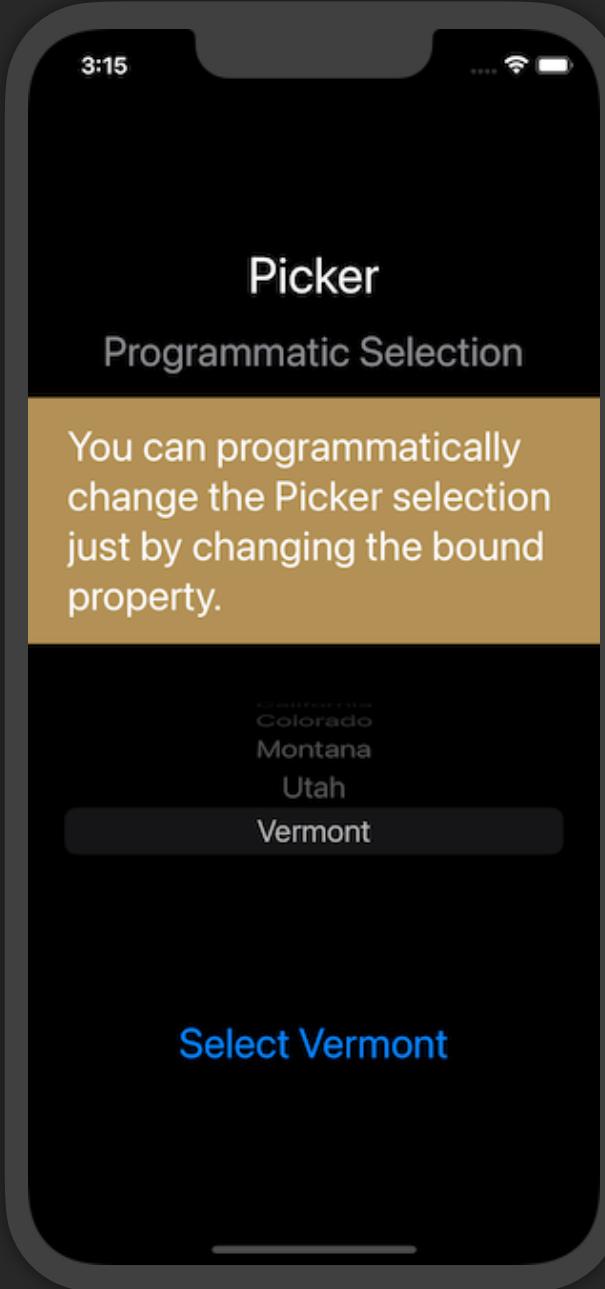


Programmatic Selection

```
struct Picker_ProgrammaticSelection: View {  
    @State private var favoriteState = 1  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Picker",  
                subtitle: "Programmatic Selection",  
                desc: "You can programmatically change the Picker selection just by  
                changing the bound property.")  
  
            Picker("States", selection: $favoriteState) {  
                Text("California").tag(0)  
                Text("Colorado").tag(1)  
                Text("Montana").tag(2)  
                Text("Utah").tag(3)  
                Text("Vermont").tag(4)  
            }  
            .pickerStyle(.wheel)  
            .padding(.horizontal)  
  
            Button("Select Vermont") {  
                withAnimation {  
                    favoriteState = 4  
                }  
            }  
            .font(.title)  
        }  
    }  
}
```

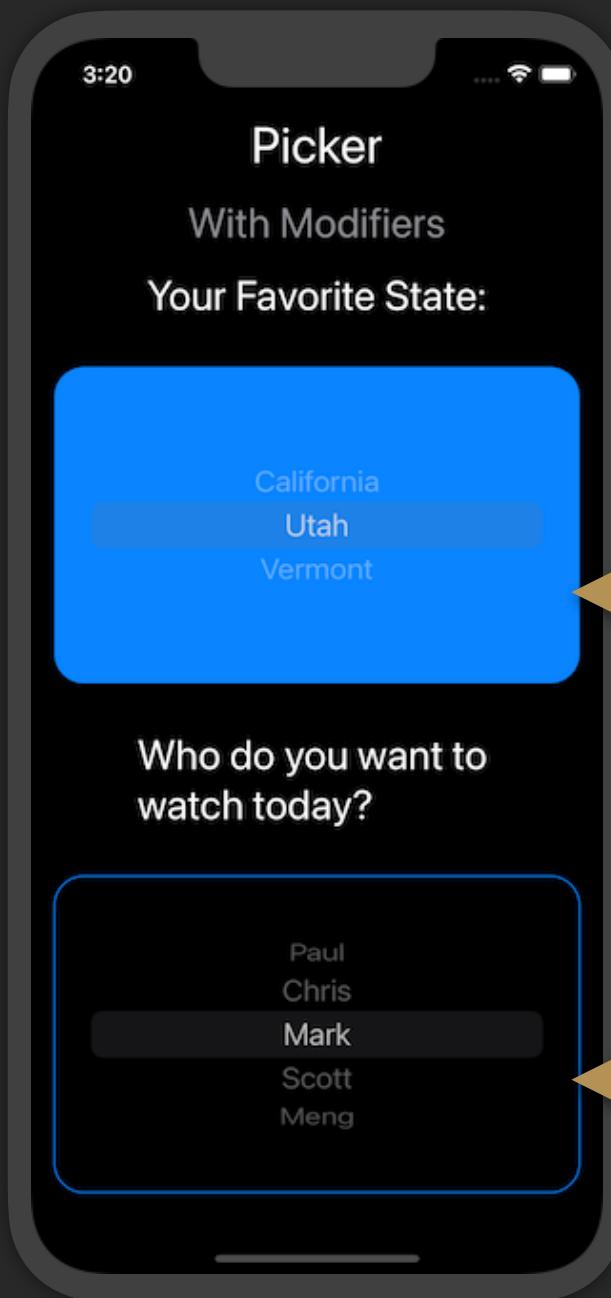
When you change the Picker's bound property value, the Picker then updates and selects the matching row.

Note: I added withAnimation so you see the wheel actually spin.





Customized



```
struct Picker_Customized : View {
    @State private var favoriteState = 1
    @State private var youTuberName = "Mark"

    var body: some View {
        VStack(spacing: 16) {
            Text("Picker").font(.largeTitle)
            Text("With Modifiers").foregroundColor(.gray)
            Text("Your Favorite State:")
            Picker("Select State", selection: $favoriteState) {
                Text("California").tag(0)
                Text("Utah").tag(1)
                Text("Vermont").tag(2)
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)
            .background(RoundedRectangle(cornerRadius: 20)
                        .fill(Color.blue))
            .padding()

            Text("Who do you want to watch today?")
            Picker("Select person", selection: $youTuberName) {
                Text("Paul").tag("Paul")
                Text("Chris").tag("Chris")
                Text("Mark").tag("Mark")
                Text("Scott").tag("Scott")
                Text("Meng").tag("Meng")
            }
            .pickerStyle(.wheel)
            .padding(.horizontal)
            .background(RoundedRectangle(cornerRadius: 20)
                        .stroke(Color.blue, lineWidth: 1))
            .padding()
        }
        .font(.title)
    }
}
```



Rows with Images

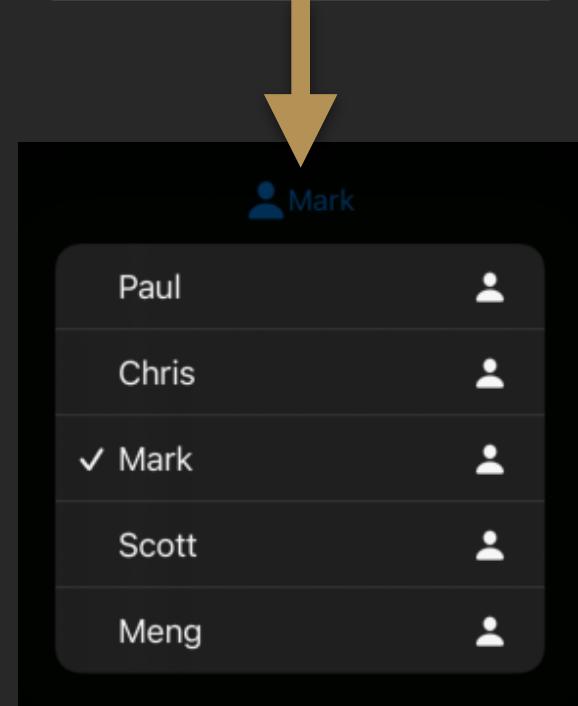
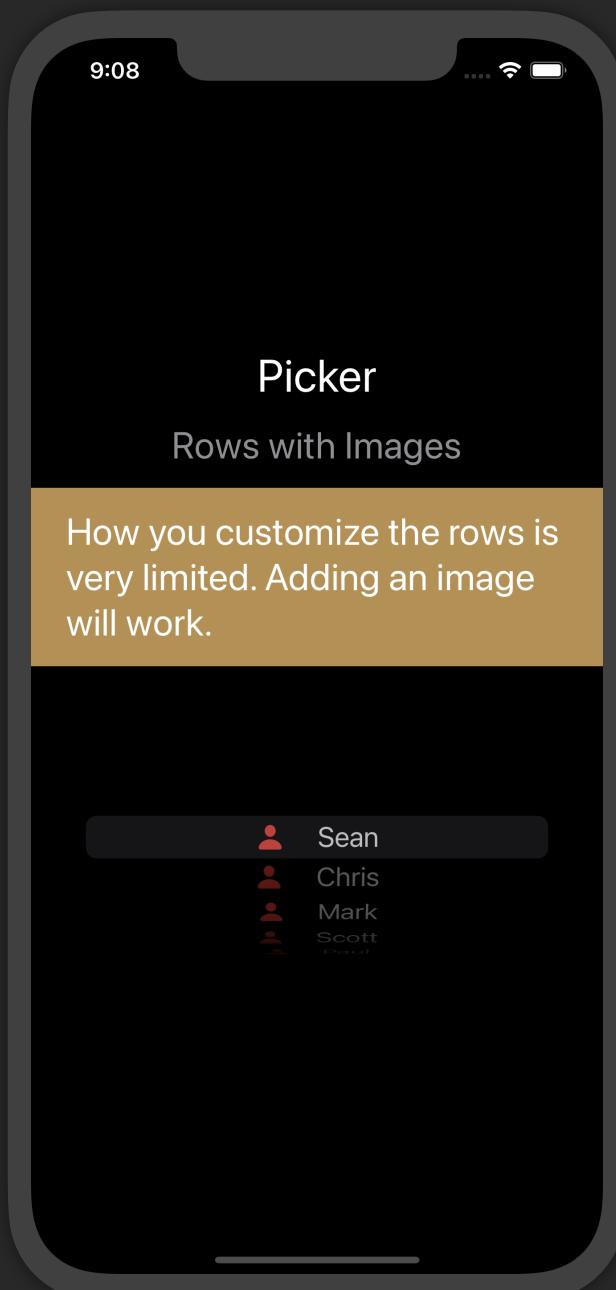
```
struct Picker_RowsWithImages : View {
    @State private var youTuberName = "Mark"

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                subtitle: "Rows with Images",
                desc: "Row customization is limited. Adding an image will work.")
            Picker(selection: $youTuberName, label: Text(""))
                .pickerStyle(.wheel)
                .padding(.horizontal)
                .font(.title)
        }
    }
}

fileprivate struct Row : View {
    var name: String

    var body: some View {
        return HStack {
            Image(systemName: "person.fill")
                .padding(.trailing)
                .foregroundColor(Color.red)
            Text(name)
        }
    }
}
```

When not using the wheel picker style, the picker looks like this with the image.





Binding Rows to Data

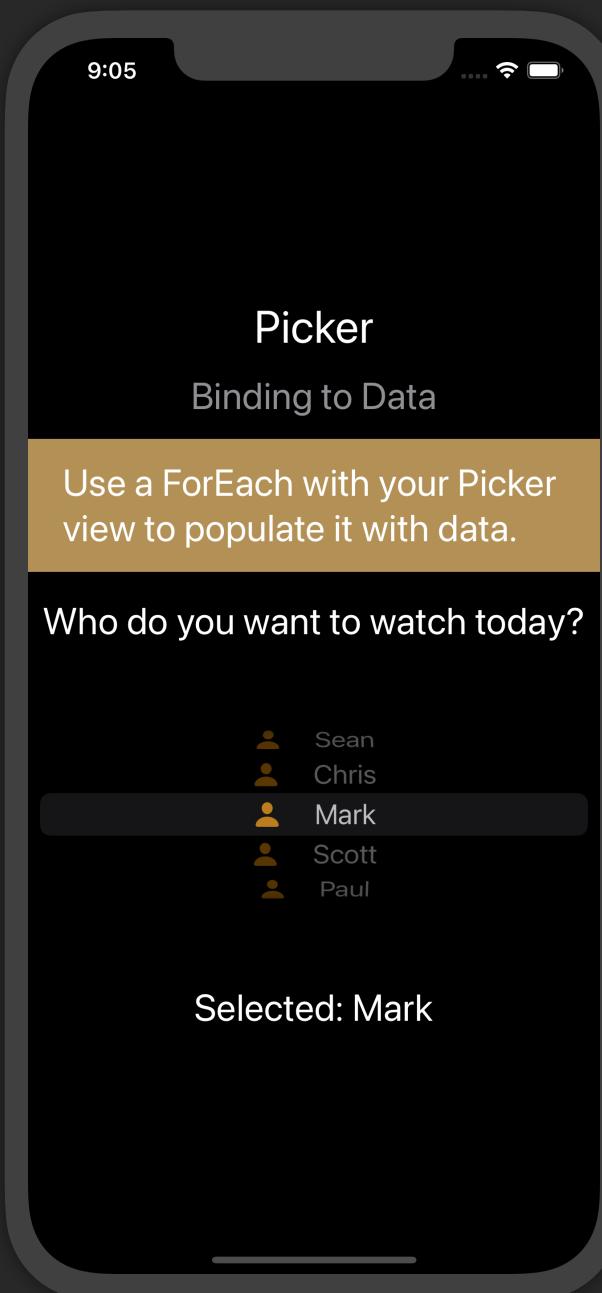
```
struct Picker_BindingToData : View {
    @State private var youTuberName = "Mark"
    var youTubers = ["Sean", "Chris", "Mark", "Scott", "Paul"]

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                subtitle: "Binding to Data",
                desc: "Use a ForEach with your Picker view to populate it with data.")

            Text("Who do you want to watch today?")

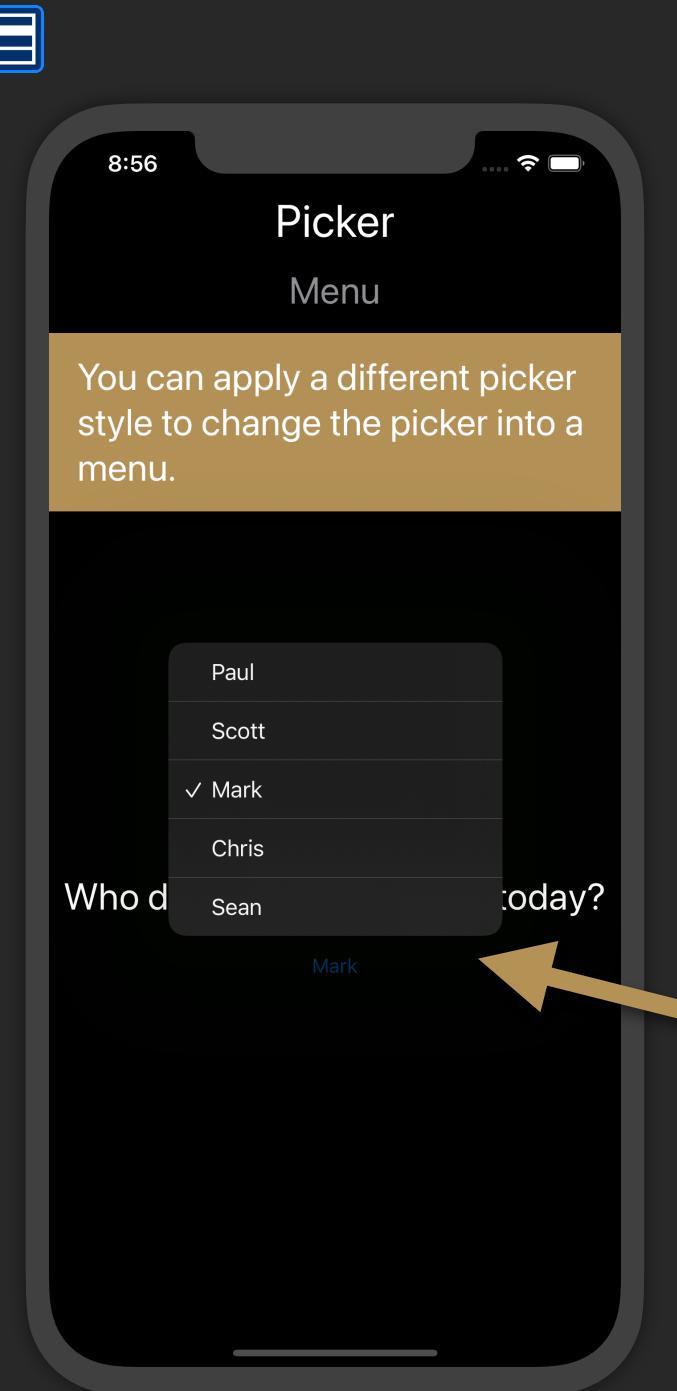
            Picker(selection: $youTuberName, label: Text(""))
                .pickerStyle(.wheel)
                .font(.title)

            fileprivate struct Row : View {
                var name: String
                var body: some View {
                    HStack {
                        Image(systemName: "person.fill")
                            .padding(.trailing)
                            .foregroundColor(Color.orange)
                        Text(name)
                    }
                }
            }
        }
    }
}
```



Menu PickerStyle

iOS 14



```

struct Picker_Menu: View {
    @State private var youTuberName = "Mark"
    var youTubers = ["Sean", "Chris", "Mark", "Scott", "Paul"]

    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Picker",
                       subtitle: "Menu",
                       desc: "You can apply a different picker style to change the picker into a menu.")

            Spacer()

            Text("Who do you want to watch today?")
                .padding(.bottom, 0)

            Picker(selection: $youTuberName, label: Text("Who do you want to watch")) {
                ForEach(youTubers, id: \.self) { name in
                    Text(name)
                }
            }
            .pickerStyle(.menu)

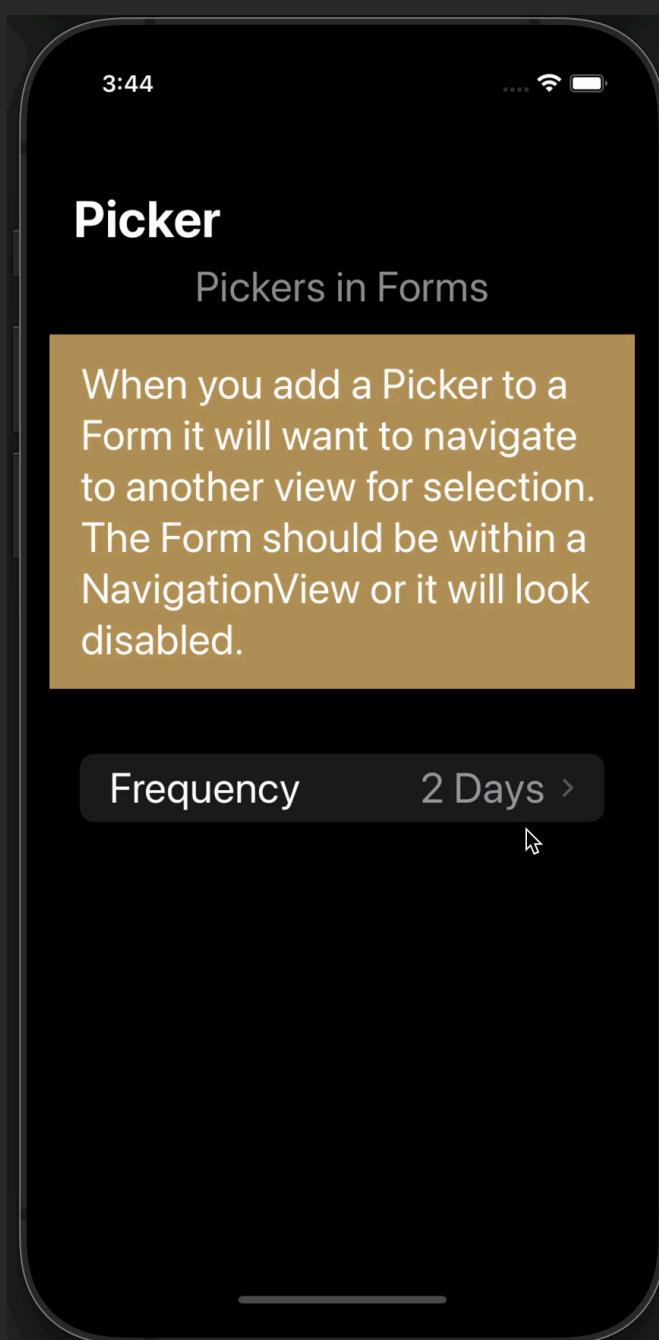
            Spacer()
        }
        .font(.title)
    }
}

```

Note: It doesn't seem we can alter the font style of the selected item from the picker at this time.

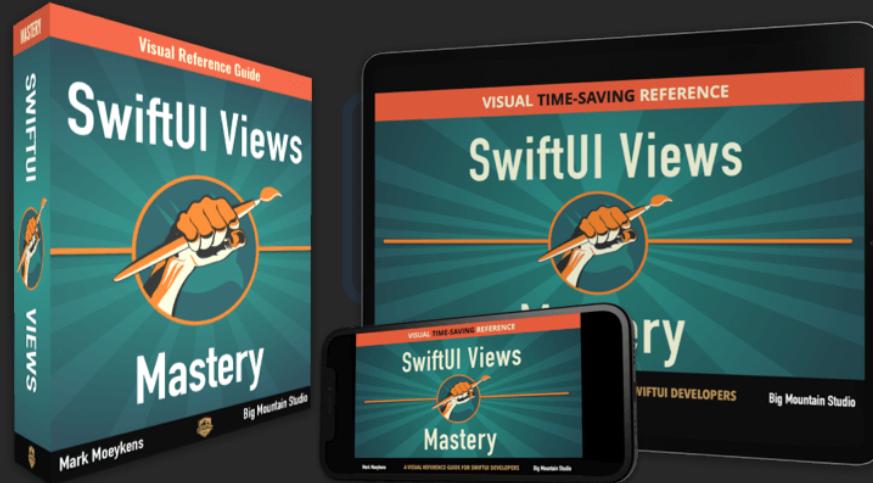
Pickers in Forms

```
struct Picker_InForm: View {  
    @State private var selectedDaysOption = "2"  
    var numberOfDaysOptions = ["1", "2", "3", "4", "5"]  
  
    var body: some View {  
        NavigationView {  
            VStack {  
                HeaderView("", subtitle: "Pickers in Forms",  
                          desc: "When you add a Picker to a Form it will want to navigate to another view for selection. The Form should be within a NavigationView or it will look disabled.")  
  
                Form {  
                    Picker("Frequency", selection: $selectedDaysOption) {  
                        ForEach(numberOfDaysOptions, id: \.self) {  
                            Text("\($0) Days").tag($0)  
                        }  
                    }  
                }  
                .navigationTitle("Picker")  
            }  
            .font(.title)  
        }  
    }  
}
```



If you don't like this behavior of Pickers in Forms (in NavigationViews) then apply a different picker style (like menu or wheel).

PhotosPicker



This SwiftUI chapter is locked in this
preview.

The photos picker first appears as a button on your UI. When tapped it will give the user access to the photos library. You can programmatically filter what you want them to see, such as just videos or time-lapse videos.

The photos picker view (the

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ProgressView



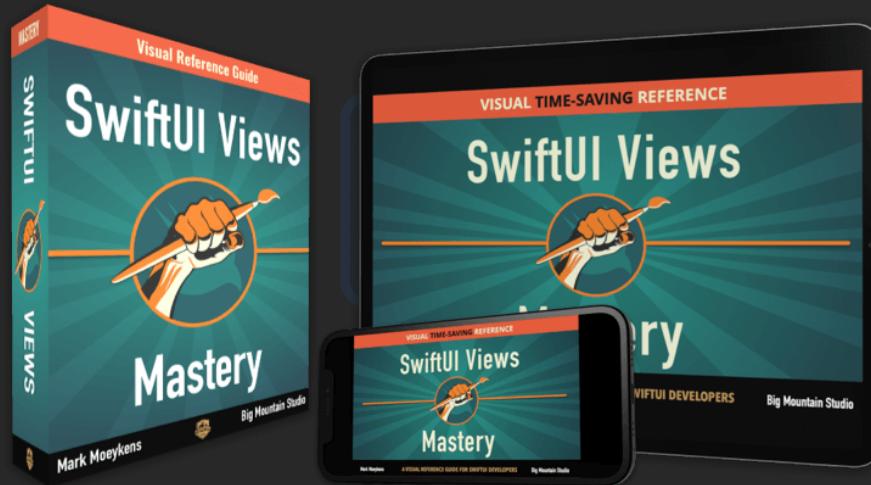
This SwiftUI chapter is locked in this

The progress view gives you three different ways to show the user that something is currently happening and optionally give you a way to show the progression of some activity.

This is a pull-in view in so

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

RenameButton



The rename button doesn't do anything by itself. You need to pair it with a `buttonAction` modifier to implement some code when the button is tapped.



This SwiftUI chapter is locked in this preview.

The button can be formatted just like the SwiftUI Button can.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ScrollView



A ScrollView is like a container for child views. When the child views within the ScrollView go outside the frame, the user can scroll to bring the child views that are outside the frame into view.

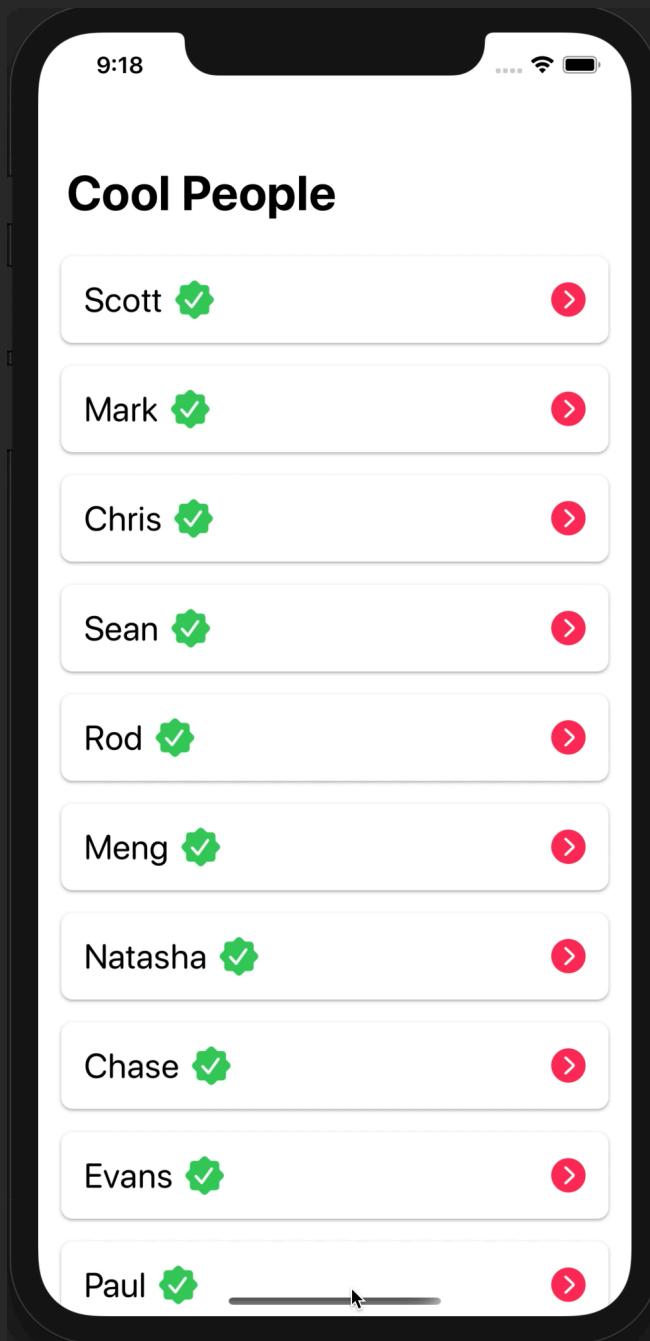
A ScrollView is a push-out view in the scroll direction you specify. You can set the direction of a ScrollView to be vertical or horizontal.

Introduction

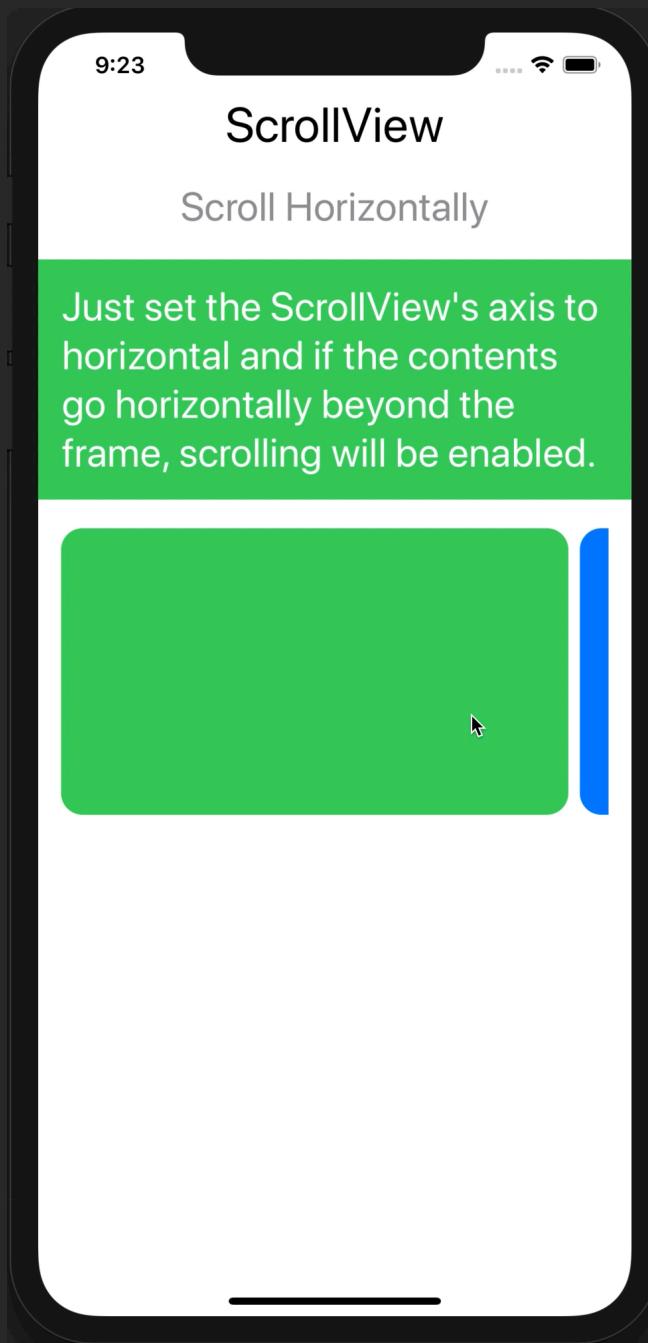
```
@State private var names = ["Scott", "Mark", "Chris", "Sean", "Rod", "Meng", "Natasha", "Chase",  
"Evans", "Paul", "Durtschi", "Max"]  
...  
NavigationView {  
    GeometryReader { gr in  
        ScrollView {  
            ForEach(self.names, id: \.self) { name in  
                NavigationLink(destination: DetailView(name: name)) {  
                    HStack {  
                        Text(name).foregroundColor(.primary)  
                        Image(systemName: "checkmark.seal.fill")  
                            .foregroundColor(.green)  
                        Spacer()  
                        Image(systemName: "chevron.right.circle.fill")  
                    }  
                    .font(.system(size: 24, design: .rounded))  
                    .padding().background(Color.white)  
                    .cornerRadius(8)  
                    .shadow(radius: 1, y: 1)  
                }  
            } // Set the width on the ForEach (it's a View)  
            .frame(width: gr.size.width - 32)  
            .tint(.pink)  
            .padding()  
        }  
        .navigationTitle(Text("Cool People"))  
    }  
}
```

Wrap the ForEach in a ScrollView.

A ScrollView with a ForEach view is similar to a List. But be warned, the rows are not reusable. It is best to limit the number of rows for memory and performance considerations.



ScrollView



```
struct ScrollView_Horizontal : View {  
    var items = [Color.green, Color.blue, Color.purple, Color.pink,  
                Color.yellow, Color.orange]  
  
    var body: some View {  
        GeometryReader { gr in  
            VStack(spacing: 20) {  
                Text("ScrollView")  
                    .font(.largeTitle)  
                Text("Scroll Horizontally")  
                    .font(.title)  
                    .foregroundColor(.gray)  
                Text("Just set the ScrollView's axis to horizontal and if the contents go  
                    horizontally beyond the frame, scrolling will be enabled.")  
                ...  
            }  
            .padding(.horizontal)  
            Spacer()  
        }  
        .background(LinearGradient(gradient: Gradient(colors: [Color.purple, Color.pink]),  
                                startPoint: .top, endPoint: .bottom))  
        .edgesIgnoringSafeArea(.all)  
    }  
}  
  
struct ScrollView_Previews: PreviewProvider {  
    static var previews: some View {  
        ScrollView_Horizontal()  
    }  
}
```

Set the scroll direction and if you want to show the scroll indicators or not.

Searchable



This SwiftUI chapter is locked in this
preview.

In iOS, the `searchable` modifier adds a text field to the `NavigationView`. Using it without a navigation view will show nothing. Use this modifier for situations where you want to give users the ability to either `search` for some data or `filter` an existing set of data already shown.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

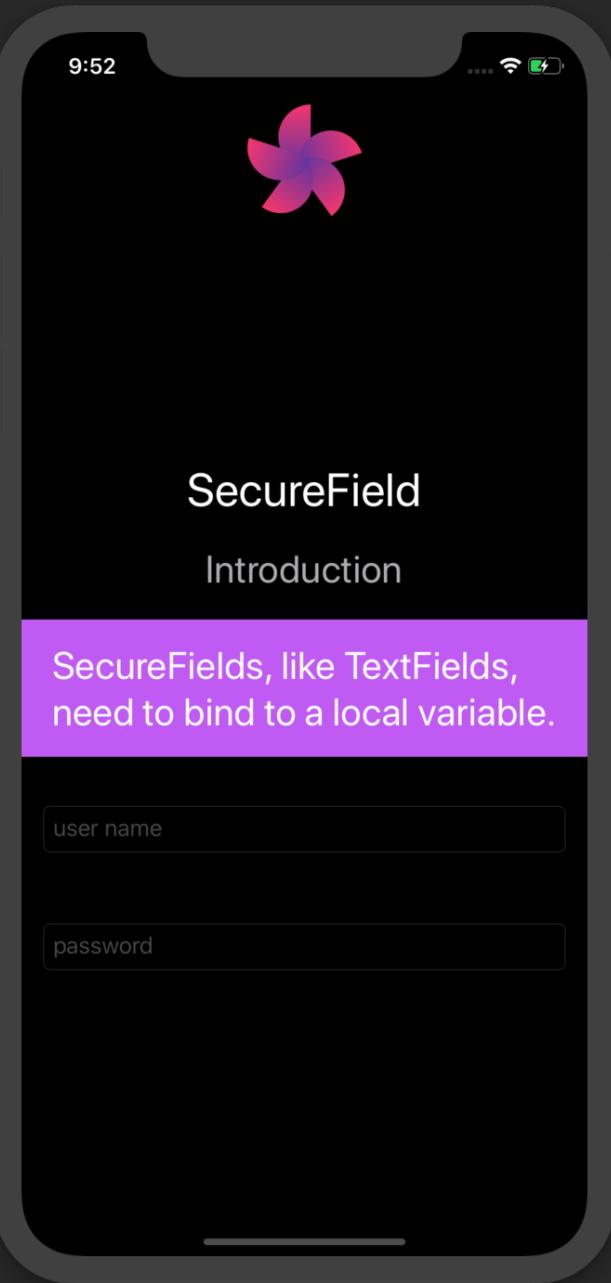
SecureField



In order to get or set the text in a SecureField, you need to bind it to a variable. This variable is passed into the SecureField's initializer. Then, all you need to do is change this bound variable's text to change what is in the SecureField. Or read the bound variable's value to see what text is currently in the SecureField.

This is a pull-in control.

Introduction



```
@State private var userName = ""  
@State private var password = ""  
...  
VStack(spacing: 20) {  
    Image("Logo")  
  
    Spacer()  
  
    Text("SecureField")  
        .font(.largeTitle)  
  
    Text("Introduction")  
        .font(.title)  
        .foregroundColor(.gray)  
  
    Text("SecureFields, like TextFields, need to bind to a local variable.")  
    ...  
  
    TextField("user name", text: $userName)  
        .textFieldStyle(.roundedBorder)  
        .padding()  
  
    SecureField("password", text: $password)  
        .textFieldStyle(.roundedBorder)  
        .padding()  
  
    Spacer()  
}
```

For textFieldStyle, use:

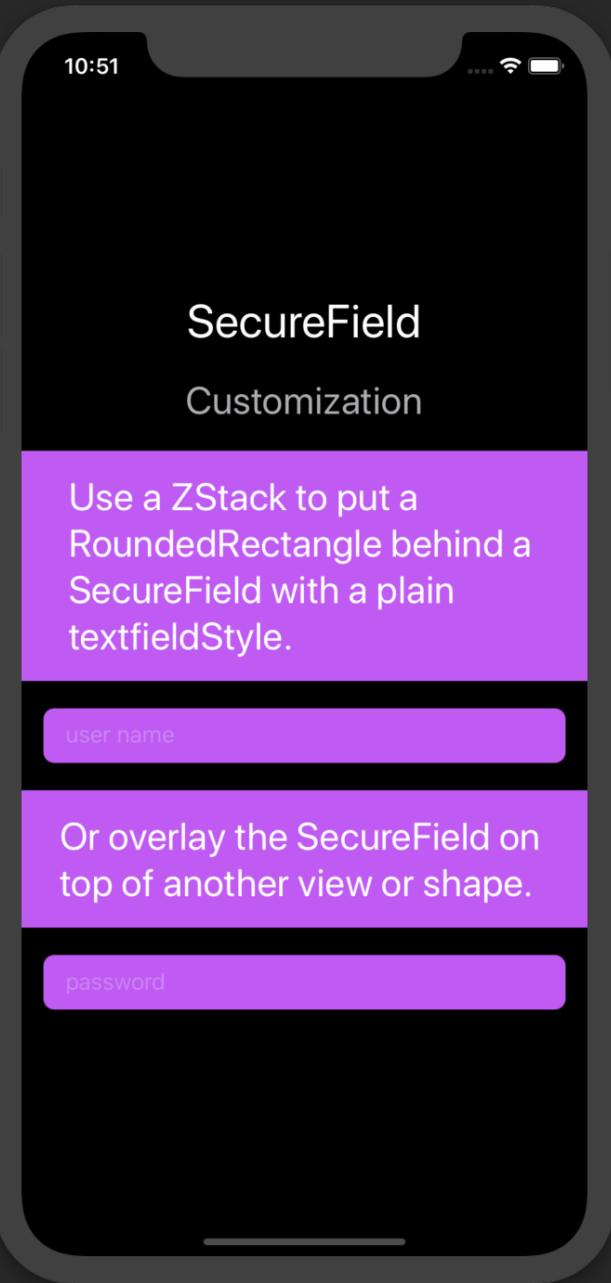
< iOS 15

RoundedBorderTextFieldStyle()

iOS 15+

.roundedBorder

Customizations



The image shows a dark-themed iPhone X mockup. At the top, the status bar displays the time '10:51' and signal strength. Below the status bar, the screen content includes a title 'SecureField Customization' and two main sections. The first section, titled 'Use a ZStack to put a RoundedRectangle behind a SecureField with a plain textFieldStyle.', contains a purple rounded rectangle background with a white SecureField labeled 'user name'. The second section, titled 'Or overlay the SecureField on top of another view or shape.', shows a white SecureField labeled 'password' overlaid on a black rounded rectangle background.

```
@State private var userName = ""
@State private var password = ""

...
```

```
Text("Use a ZStack to put a RoundedRectangle behind a SecureField with a plain textFieldStyle.")
```

```
...
```

```
ZStack{
    RoundedRectangle(cornerRadius: 8)
        .foregroundColor(.purple)
    TextField("user name", text: $userName)
        .foregroundColor(Color.white)
        .padding(.horizontal)
}
.frame(height: 40)
.padding(.horizontal)
```

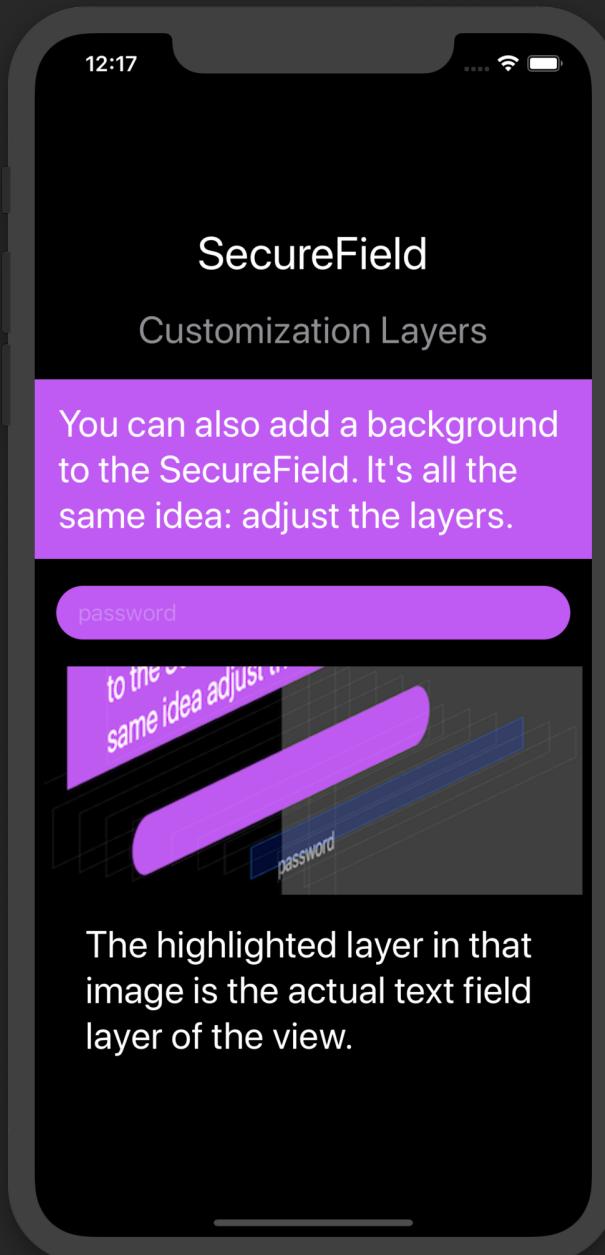
```
Text("Or overlay the SecureField on top of another view or shape.")
```

```
...
```

```
RoundedRectangle(cornerRadius: 8)
    .foregroundColor(.purple)
    .overlay(
        SecureField("password", text: $password)
            .foregroundColor(Color.white)
            .padding(.horizontal)
    )
    .frame(height: 40)
    .padding(.horizontal)
```

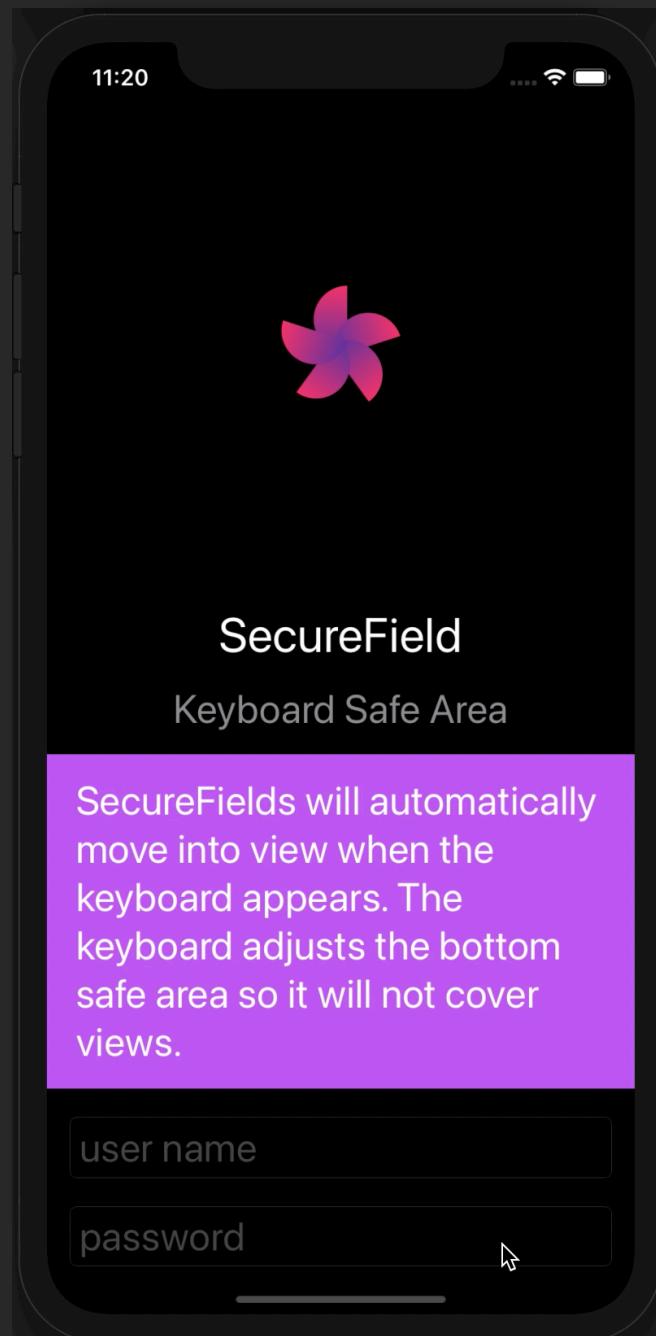
Customization Layers

```
@State private var userName = ""  
@State private var password = ""  
...  
VStack(spacing: 20) {  
    Text("SecureField")  
        .font(.largeTitle)  
    Text("Customization Layers")  
        .font(.title)  
        .foregroundColor(.gray)  
    Text("You can also add a background to the SecureField. It's all the same idea: adjust the layers.")  
    ...  
  
    SecureField("password", text: $password)  
        .foregroundColor(Color.white)  
        .frame(height: 40)  
        .padding(.horizontal)  
        .background(  
            Capsule()  
                .foregroundColor(.purple)  
        )  
        .padding(.horizontal)  
  
    Image("SecureFieldLayers")  
  
    Text("The highlighted layer in that image is the actual text field layer of the view.")  
        .font(.title)  
        .padding(.horizontal)  
}
```



Keyboard Safe Area

iOS 14



```
struct SecureField_KeyboardSafeArea: View {  
    @State private var userName = ""  
    @State private var password = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Spacer()  
            Image("Logo")  
            Spacer()  
  
            HeaderView("SecureField",  
                      subtitle: "Keyboard Safe Area",  
                      desc: "SecureFields will automatically move into view when the keyboard  
appears. The keyboard adjusts the bottom safe area so it will not cover views.",  
                      back: .purple, textColor: .white)  
  
            TextField("user name", text: $userName)  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
  
            SecureField("password", text: $password)  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
        }  
        .font(.title)  
    }  
}
```

Segmented Control (Picker)



Segmented controls are now Picker controls with a different picker style set. In order to get or set the selected segment, you need to bind it to a variable. This variable is passed into the segmented control's (Picker's) initializer. Then, all you need to do is change this bound variable's value to change the selected segment. Or read the bound variable's value to see which segment is currently selected.

This is a pull-in view.



Introduction

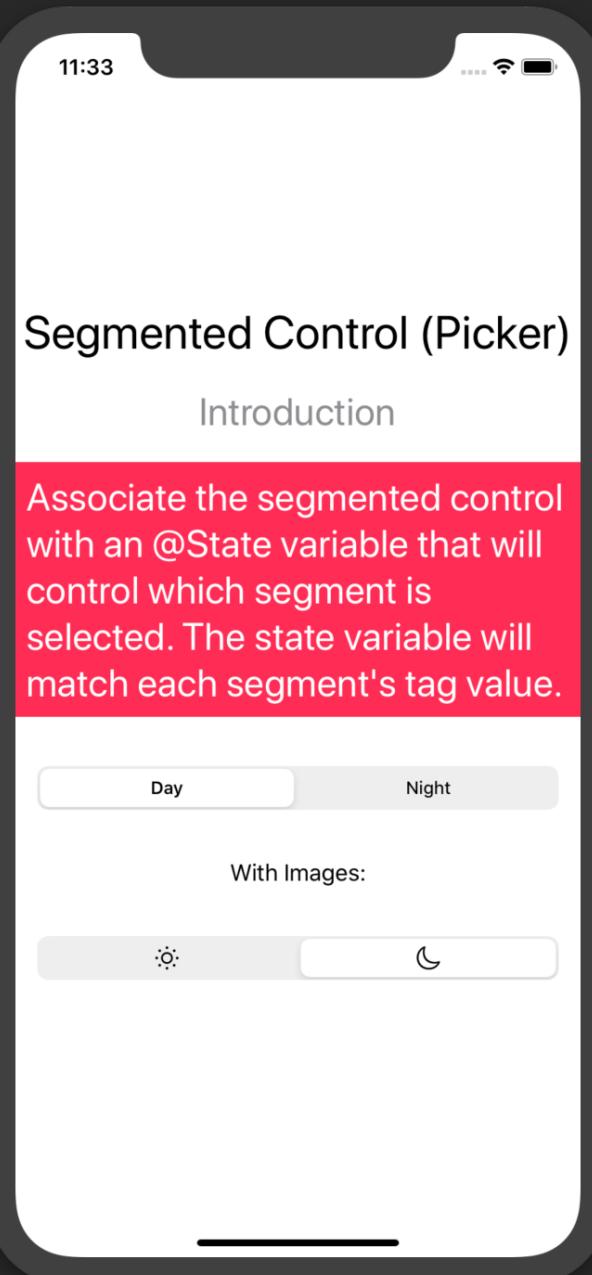
```
@State private var dayNight = "day"
@State private var tab = 1

...
VStack(spacing: 20) {
    Text("Segmented Control (Picker)").font(.largeTitle)
    Text("Introduction")
        .font(.title).foregroundColor(.gray)
    Text("Associate the segmented control with an @State variable that will control which segment is selected. The state variable will match each segment's tag value.")
}

...
Picker("", selection: $dayNight) {
    Text("Day").tag("day")
    Text("Night").tag("night")
}
    .pickerStyle(.segmented)
    .padding()

Text("With Images:")

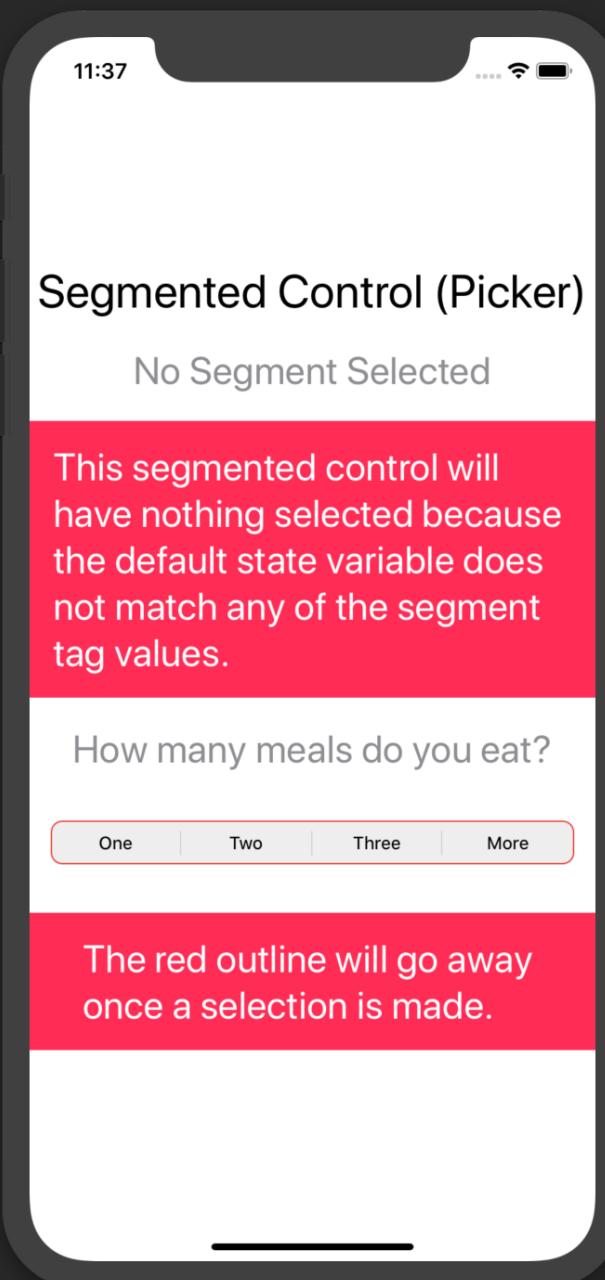
Picker("", selection: $tab) {
    Image(systemName: "sun.min").tag(0)
    Image(systemName: "moon").tag(1)
}
    .pickerStyle(.segmented)
    .padding()
}
```





No Segment Selected

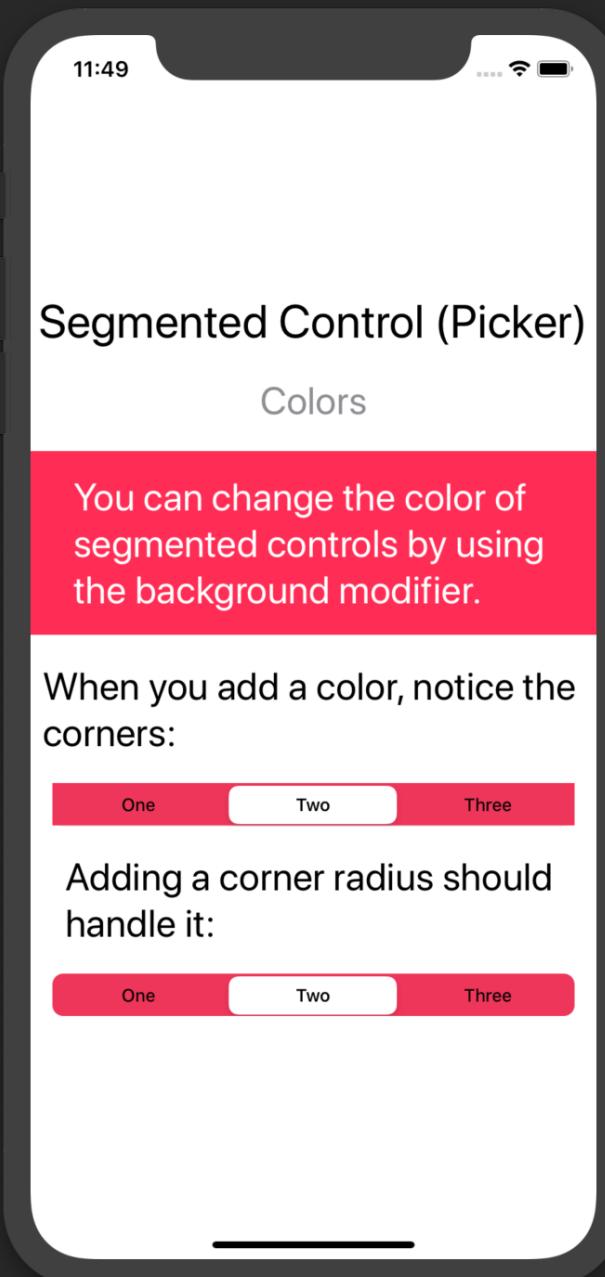
```
@State private var selection = 0  
...  
VStack(spacing: 20) {  
    Text("Segmented Control (Picker)").font(.largeTitle)  
    Text("No Segment Selected")  
        .font(.title).foregroundColor(.gray)  
    Text("This segmented control will have nothing selected because the default state variable  
does not match any of the segment tag values.")  
    ...  
    Text("How many meals do you eat?")  
        .foregroundColor(.gray)  
        .font(.title)  
    Picker("", selection: $selection) {  
        Text("One").tag(1)  
        Text("Two").tag(2)  
        Text("Three").tag(3)  
        Text("More").tag(4)  
    }  
        .pickerStyle(.segmented)  
        .background(RoundedRectangle(cornerRadius: 8)  
            .stroke(Color.red, lineWidth: selection == 0 ? 1 : 0))  
        .padding()  
    Text("The red outline will go away once a selection is made.")  
    ...  
}
```



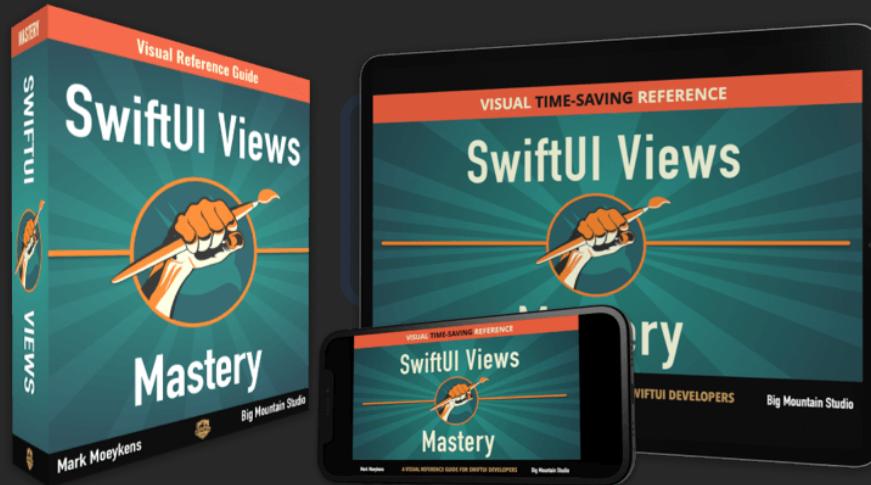
Colors

```
@State private var selection = 2
...
VStack(spacing: 20) {
    Text("Segmented Control (Picker)").font(.largeTitle)
    Text("Colors")
        .font(.title).foregroundColor(.gray)
    Text("You can change the color of segmented controls by using the background modifier.")
        ...
    Text("When you add a color, notice the corners:")
    Picker("", selection: $selection) {
        Text("One").tag(1)
        Text("Two").tag(2)
        Text("Three").tag(3)
    }
    .pickerStyle(.segmented)
    .background(Color.pink)
    .padding(.horizontal)

    Text("Adding a corner radius should handle it:")
    Picker("", selection: $selection) {
        Text("One").tag(1)
        Text("Two").tag(2)
        Text("Three").tag(3)
    }
    .pickerStyle(.segmented)
    .background(Color.pink)
    .cornerRadius(8)
    .padding(.horizontal)
}
```



ShareLink



This SwiftUI chapter is locked in this
preview.

The share link button gives you a convenient way to for users to share some text, a URL, collections of data, photos, etc.

It can be styled and customized just like a button.

This is a pull-in view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

SignInWithAppleButton



This SwiftUI chapter is locked in this
preview.

Apple provides you with a **SignInWithAppleButton** that you use in your app to assist your users when it comes time to implementing signing in with Apple.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

Slider



When using a Slider view, the default range of values is 0.0 to 1.0. You bind the Slider to a state variable, usually a number type, like an Int. But it doesn't have to be a number type. It can be any type that conforms to the Stridable protocol. ("Stride" means to "take steps in a direction; usually long steps".) A type that conforms to Stridable (such as an Int) means it has values that are continuous and can be stepped through and measured. ("Step through", "Stride", I think you see the connection now.)

You use the bound variable to set or get the value the Slider's thumb (circle) is currently at.

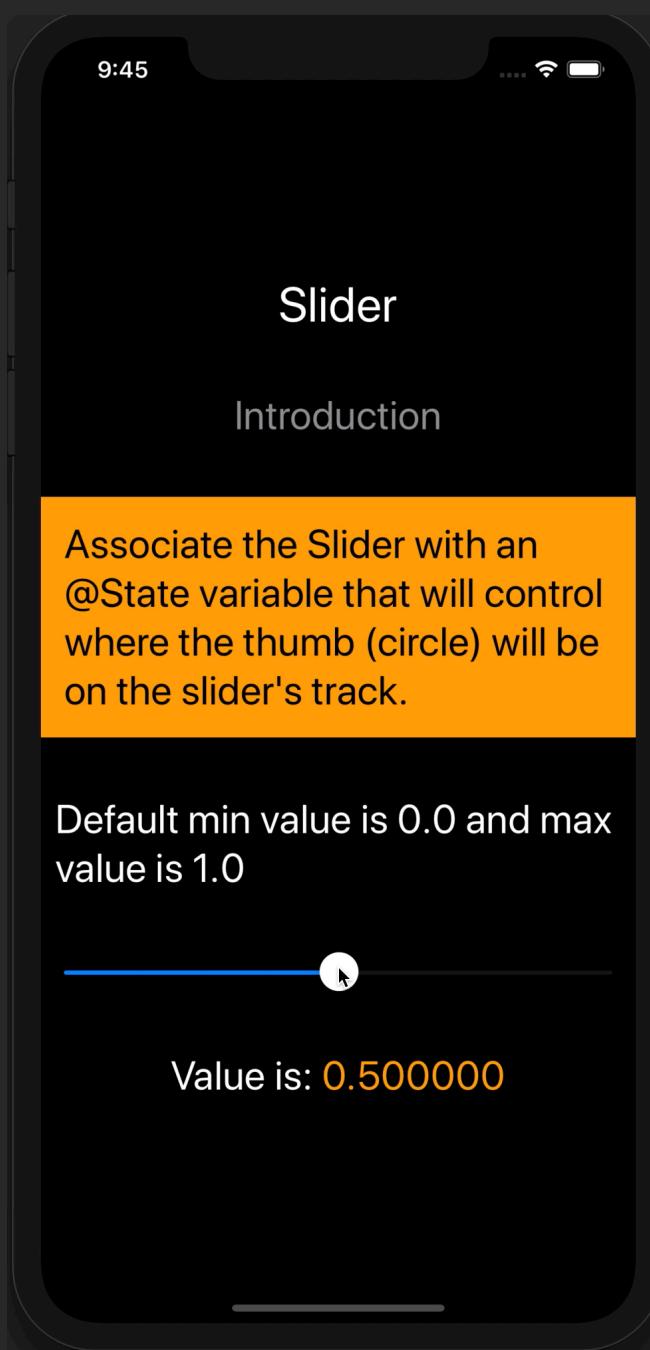
This is a pull-in view.

Introduction

```
struct Slider_Intro : View {  
    @State private var sliderValue = 0.5  
  
    var body: some View {  
        VStack(spacing: 40) {  
            Text("Slider").font(.largeTitle)  
            Text("Introduction").foregroundColor(.gray)  
            Text("Associate the Slider with an @State variable that will control where the thumb  
(circle) will be on the slider's track.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.orange).foregroundColor(Color.black)  
  
            Text("Default min value is 0.0 and max value is 1.0")  
  
            Slider(value: $sliderValue)  
                .padding(.horizontal)  
  
            Text("Value is: ") +  
            Text("\(sliderValue)").foregroundColor(.orange)  
        }.font(.title)  
    }  
}
```

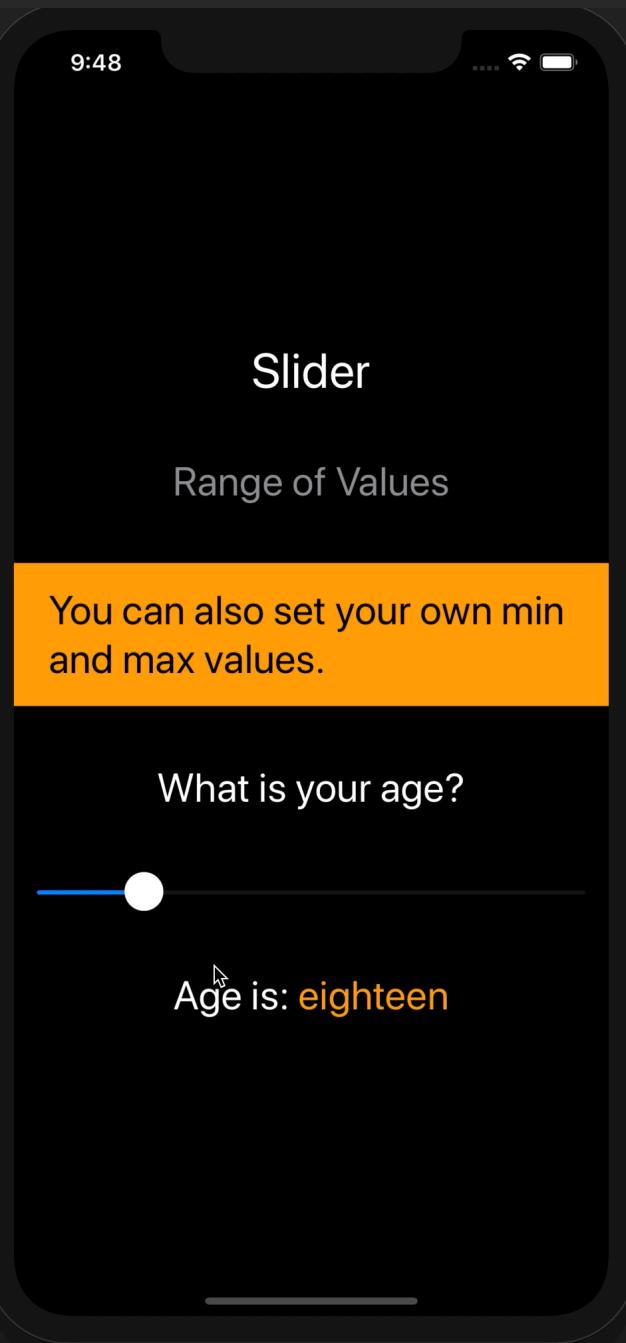
Value used for the slider.

Set the state variable in the slider's initializer.





Range of Values (Minimum & Maximum)



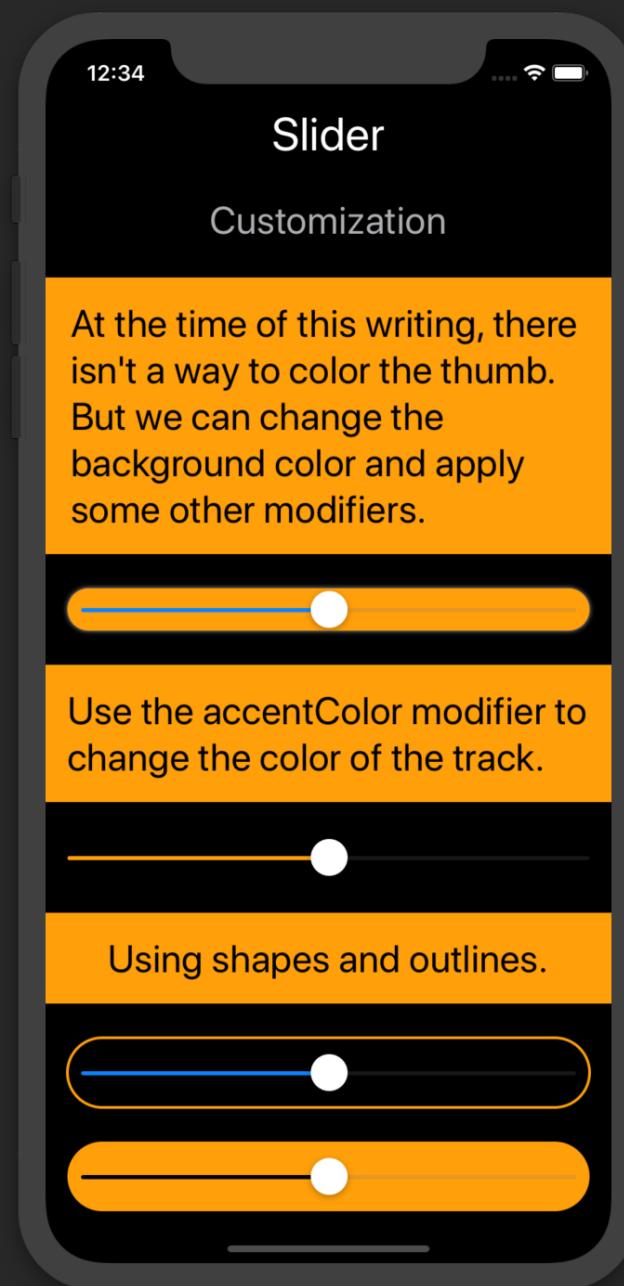
```
struct Slider_RangeOfValues: View {  
    @State private var age = 18.0  
  
    let ageFormatter: NumberFormatter = {  
        let numFormatter = NumberFormatter()  
        numFormatter.numberStyle = .spellOut  
        return numFormatter  
    }()  
  
    var body: some View {  
        VStack(spacing: 40) {  
            Text("Slider").font(.largeTitle)  
            Text("Range of Values").foregroundColor(.gray)  
            Text("You can also set your own min and max values.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.orange).foregroundColor(Color.black)  
  
            Text("What is your age?")  
            Slider(value: $age, in: 1...100, step: 1)  
                .padding(.horizontal)  
  
            Text("Age is: ") +  
            Text("\(ageFormatter.string(from: NSNumber(value: age))!)")  
                .foregroundColor(.orange)  
        }.font(.title)  
    }  
}
```

Format the slider value into a spelled-out number.

Provide a range here.

The step parameter defines the increment from one value to the next.

Customization



```
@State private var sliderValue = 0.5
...
Text("At the time of this writing, there isn't a way to color the thumb. But we can change the
background color and apply some other modifiers.")

...
Slider(value: $sliderValue)
    .padding(.horizontal, 10)
    .background(Color.orange)
    .shadow(color: .gray, radius: 2)
    .padding(.horizontal)

Text("Use the accentColor modifier to change the color of the track.")
...

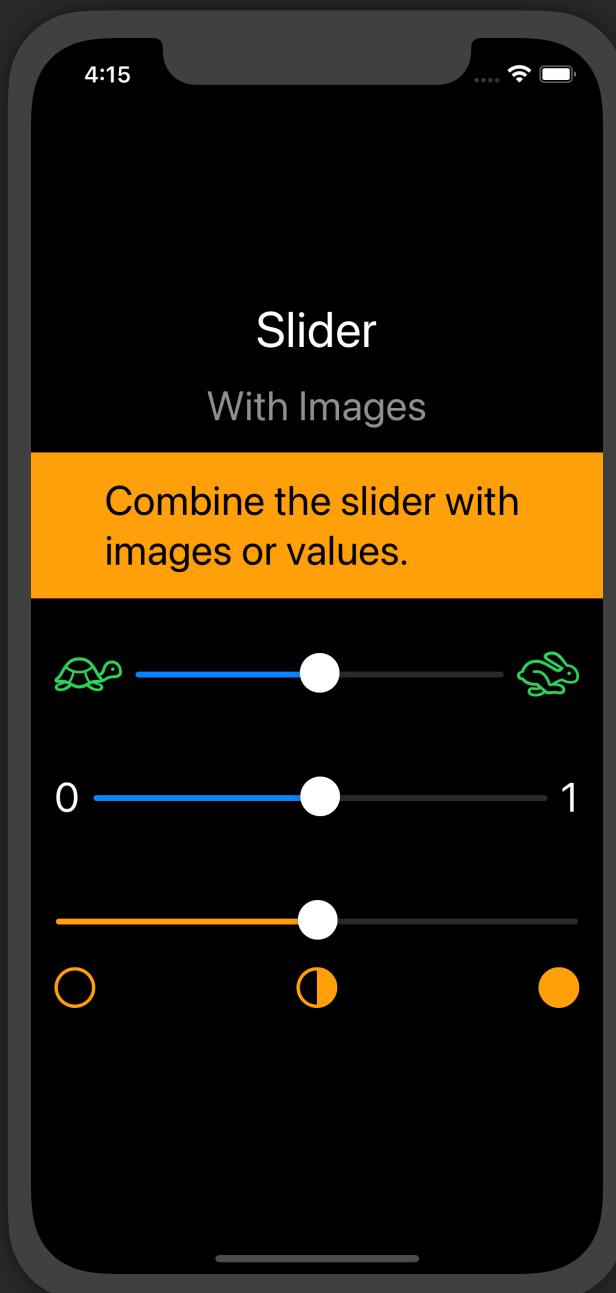
Slider(value: $sliderValue)
    .padding(.horizontal)
    .accentColor(.orange)

Text("Using shapes and outlines.")

...
Slider(value: $sliderValue)
    .padding(10)
    .background(Capsule().stroke(Color.orange, lineWidth: 2))
    .padding(.horizontal)

Slider(value: $sliderValue)
    .padding(10)
    .background(Capsule().fill(Color.orange))
    .accentColor(.black)
    .padding(.horizontal)
```

With Images



```
struct Slider_WithImages : View {
    @State private var sliderValue = 0.5

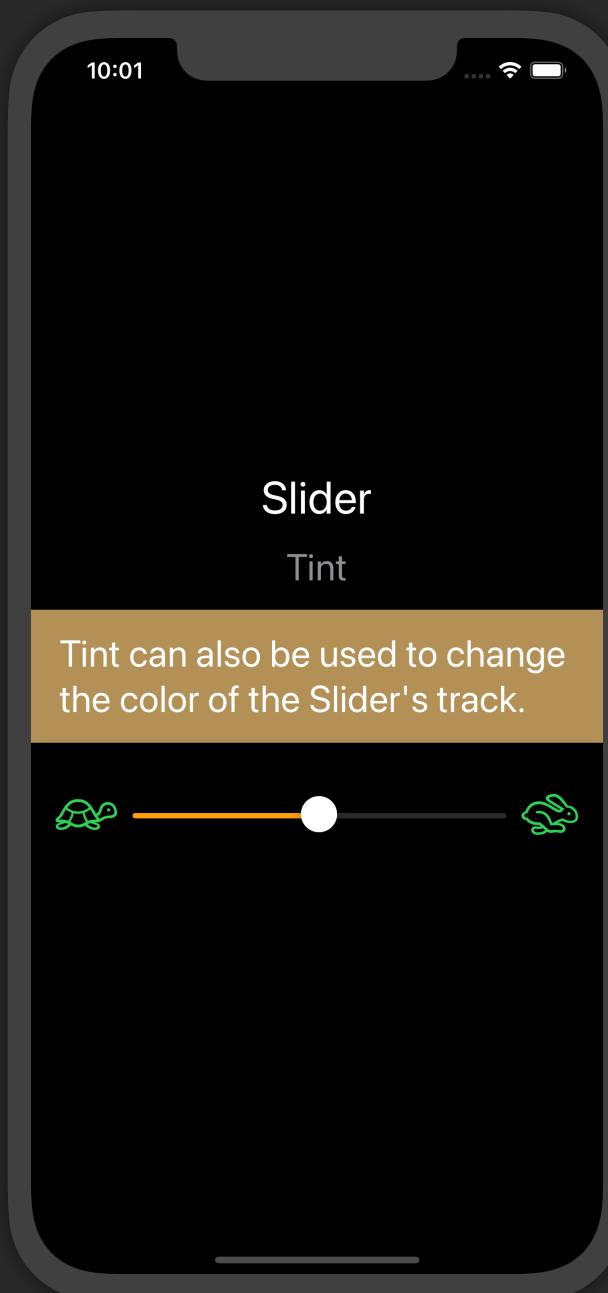
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Slider",
                subtitle: "With Images",
                desc: "Combine the slider with images or values.",
                back: .orange, textColor: .black)

            Slider(value: $sliderValue,
                minimumValueLabel: Image(systemName: "tortoise"),
                maximumValueLabel: Image(systemName: "hare"), label: {})
                .foregroundColor(.green)
                .padding()

            Slider(value: $sliderValue,
                minimumValueLabel: Text("0"),
                maximumValueLabel: Text("1"), label: {})
                .padding()

            VStack {
                Slider(value: $sliderValue)
                    .accentColor(.orange)
                HStack {
                    Image(systemName: "circle")
                    Spacer()
                    Image(systemName: "circle.righthalf.fill")
                    Spacer()
                    Image(systemName: "circle.fill")
                }
                .foregroundColor(.orange)
                .padding(.top, 8)
            }.padding()
            .font(.title)
        }
    }
}
```

Use minimum and maximum value labels to add text or images to the ends of the slider.



Tint

iOS 15



```
struct Slider_Tint: View {  
    @State private var sliderValue = 0.5  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("Slider",  
                      subtitle: "Tint",  
                      desc: "Tint can also be used to change the color of the Slider's track.")  
  
            Slider(value: $sliderValue,  
                   minimumValueLabel: Image(systemName: "tortoise"),  
                   maximumValueLabel: Image(systemName: "hare"), label: {})  
                .foregroundColor(.green)  
                .tint(.orange) ←  
                .padding()  
        }  
        .font(.title)  
    }  
}
```

Stepper

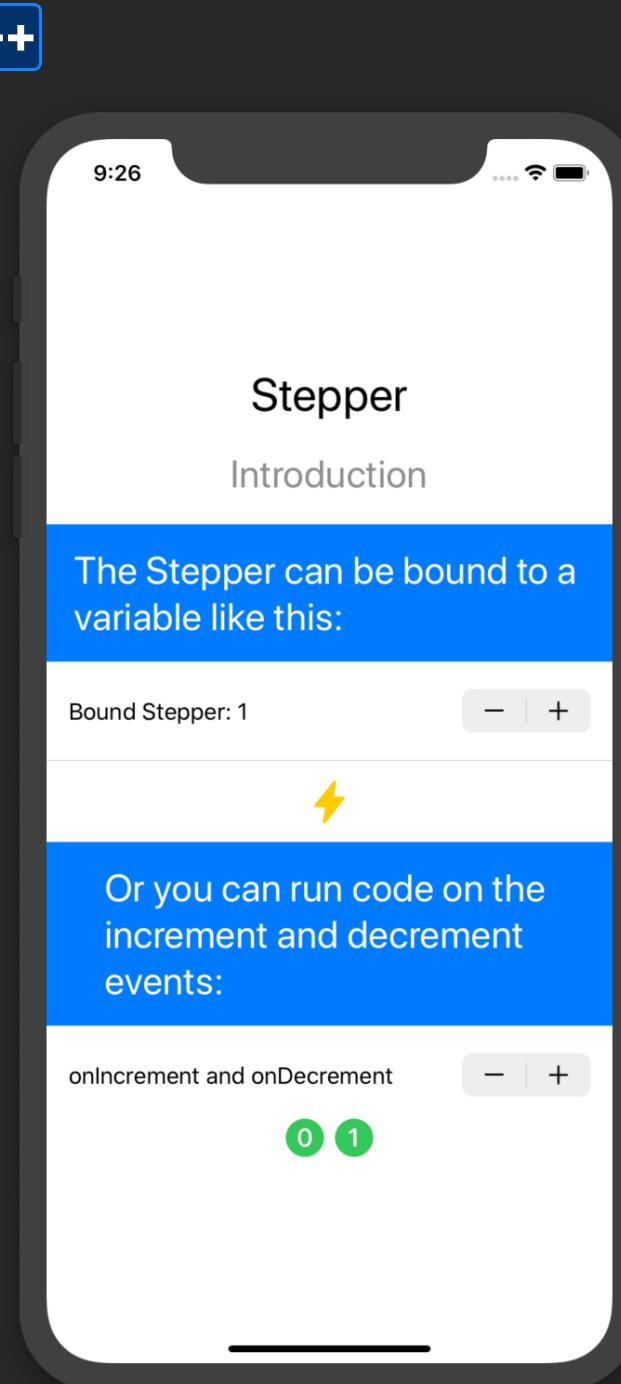


When using a Stepper view, you bind it to a state variable, usually a number. But it doesn't have to be a number type. It can be any type that conforms to the Stridable protocol. ("Stride" means to "take steps in a direction; usually long steps".) A type that conforms to Stridable means it has values that are continuous and can be stepped through and measured. ("Step through", "Stride", I think you see the connection now.)

You use the bound variable to set or get the value it is currently at.

This is a horizontal push-out view. Vertically it is pull-in.

Introduction



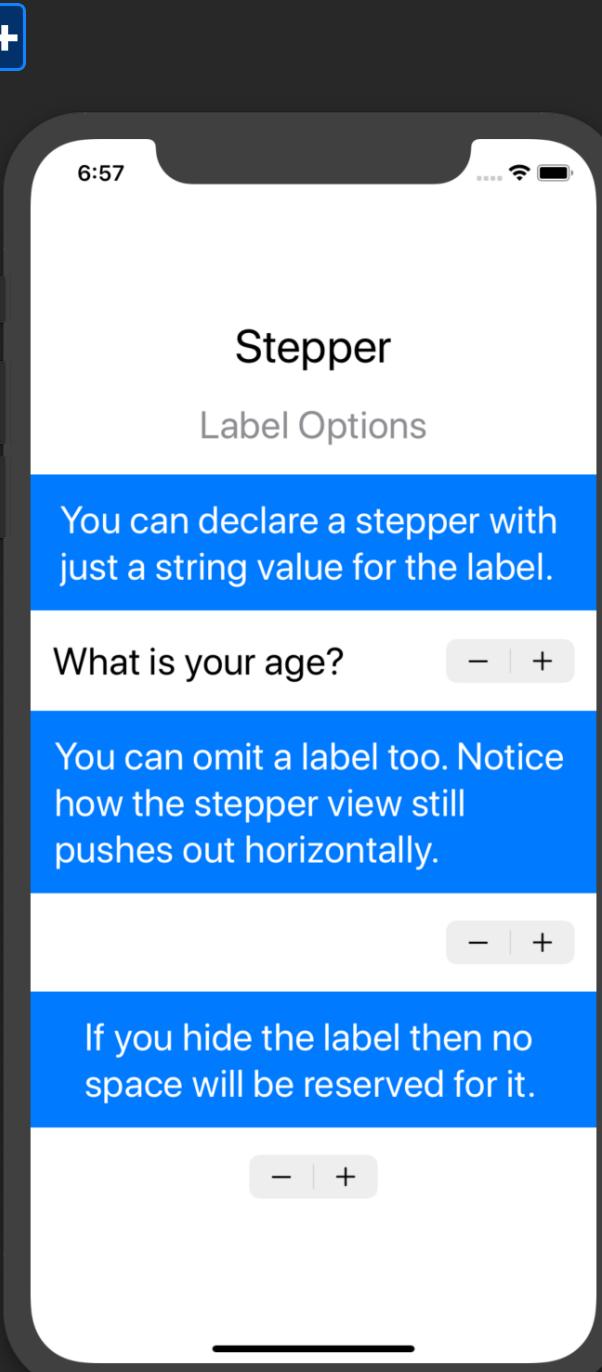
```
@State private var stepperValue = 1
@State private var values = [0, 1]
...
VStack(spacing: 20) {
    Text("Stepper")
        .font(.largeTitle)
    Text("Introduction")
        .font(.title).foregroundColor(.gray)
    Text("The Stepper can be bound to a variable like this:")
    ...
}

Stepper(value: $stepperValue) {
    Text("Bound Stepper: \(stepperValue)")
}.padding(.horizontal)
Divider()
Image(systemName: "bolt.fill")
    .font(.title).foregroundColor(.yellow)
Text("Or you can run code on the increment and decrement events:")
    .frame(maxWidth: .infinity).padding()
    .background(Color.blue).foregroundColor(Color.white)
    .font(.title)
Stepper(onIncrement: {self.values.append(self.values.count)},
       onDecrement: {self.values.removeLast()}) {
    Text("onIncrement and onDecrement")
}.padding(.horizontal)
HStack {
    ForEach(values, id: \.self) { value in
        Image(systemName: "\((value).circle.fill")
    }
}.font(.title).foregroundColor(.green)
}
```

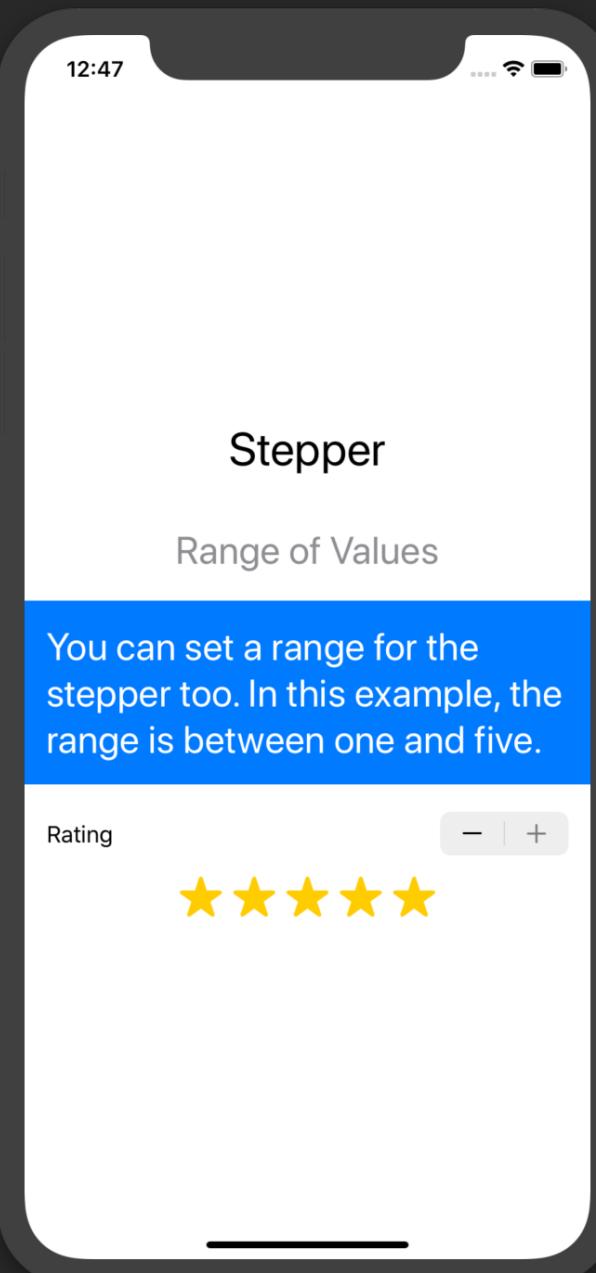
Label Options

```
struct Stepper_LabelsHidden: View {  
    @State private var stepperValue = 1  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Stepper").font(.largeTitle)  
            Text("Label Options").foregroundColor(.gray)  
            Text("You can declare a stepper with just a string value for the label.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.blue).foregroundColor(Color.white)  
            Stepper("What is your age?", value: $stepperValue)  
                .padding(.horizontal)  
            Text("You can omit a label too. Notice how the stepper view still pushes out  
horizontally.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.blue).foregroundColor(Color.white)  
            Stepper("", value: $stepperValue)  
                .padding(.horizontal)  
            Text("If you hide the label then no space will be reserved for it.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.blue).foregroundColor(Color.white)  
            Stepper("What is your age?", value: $stepperValue)  
                .padding(.horizontal)  
                .labelsHidden()  
        }.font(.title)  
    }  
}
```

Note: Even though the label/title is not shown, I would still recommend having one because it will still be used for accessibility purposes.



Range



```
@State private var stars = 0

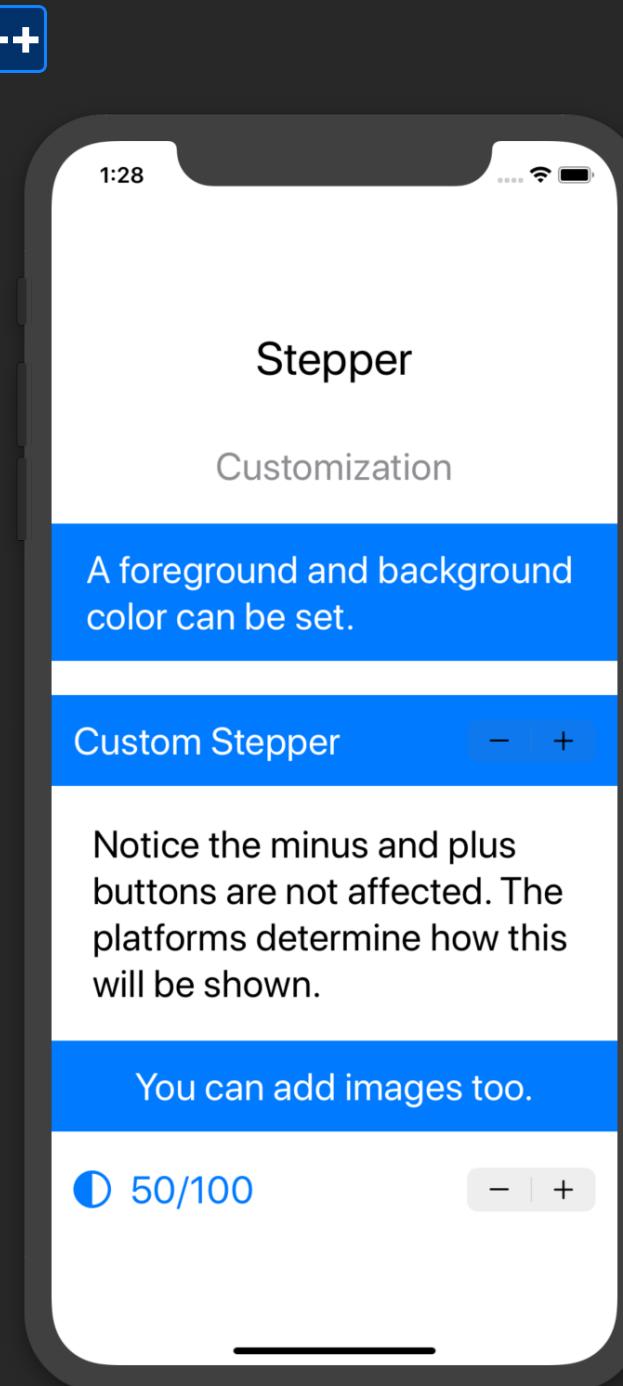
VStack(spacing: 20) {
    Text("Stepper")
        .font(.largeTitle)
        .padding()
    Text("Range of Values")
        .font(.title)
        .foregroundColor(.gray)
    Text("You can set a range for the stepper too. In this example, the range is between one and five.")
    ...
}

Stepper(value: $stars, in: 1...5) {
    Text("Rating")
}.padding(.horizontal)

HStack {
    ForEach(1...stars, id: \.self) { star in
        Image(systemName: "star.fill")
    }
    .font(.title)
    .foregroundColor(.yellow)
}
```

When the Stepper reaches the range limits, the corresponding plus or minus button will appear as disabled. In this screenshot, notice the plus button is disabled.

Customization



```
@State private var contrast = 50
...
Text("A foreground and background color can be set.")
...

Stepper(onIncrement: {}, onDecrement: {}) {
    Text("Custom Stepper")
        .font(.title)
        .padding(.vertical)
}
.padding(.horizontal)
.background(Color.blue)
.foregroundColor(.white)

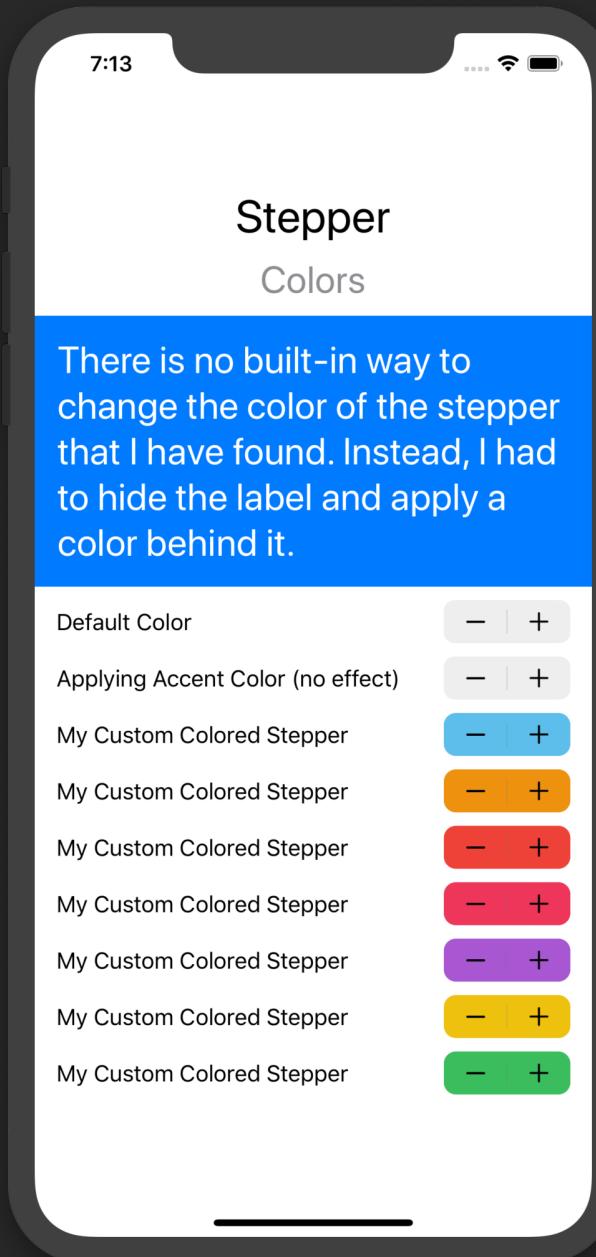
Text("Notice the minus and plus buttons are not affected. The platforms determine how this will be shown.")

...

Text("You can add images too.")
    .frame(maxWidth: .infinity).padding()
    .background(Color.blue).foregroundColor(Color.white)
    .font(.title)

Stepper(value: $contrast, in: 0...100) {
    // SwiftUI implicitly uses an HStack here
    Image(systemName: "circle.lefthalf.fill")
    Text("\(contrast)/100")
}
.font(.title)
.padding(.horizontal)
.foregroundColor(.blue)
```

Colors



Text("There is no built-in way to change the color of the stepper that I have found. Instead, I had to hide the label and apply a color behind it.")
....

```
Stepper(value: $contrast, in: 0...100) {
    Text("Applying Accent Color (no effect)")
}
.accentColor(.blue)

HStack {
    Text("My Custom Colored Stepper")
    Spacer()
    Stepper("", value: $contrast)
        .background(Color(UIColor.systemTeal))
        .cornerRadius(9)
        .labelsHidden() // Hide the label
}

HStack {
    Text("My Custom Colored Stepper")
    Spacer()
    Stepper("", value: $contrast)
        .background(Color.orange)
        .cornerRadius(9)
        .labelsHidden() // Hide the label
}
```

TabView

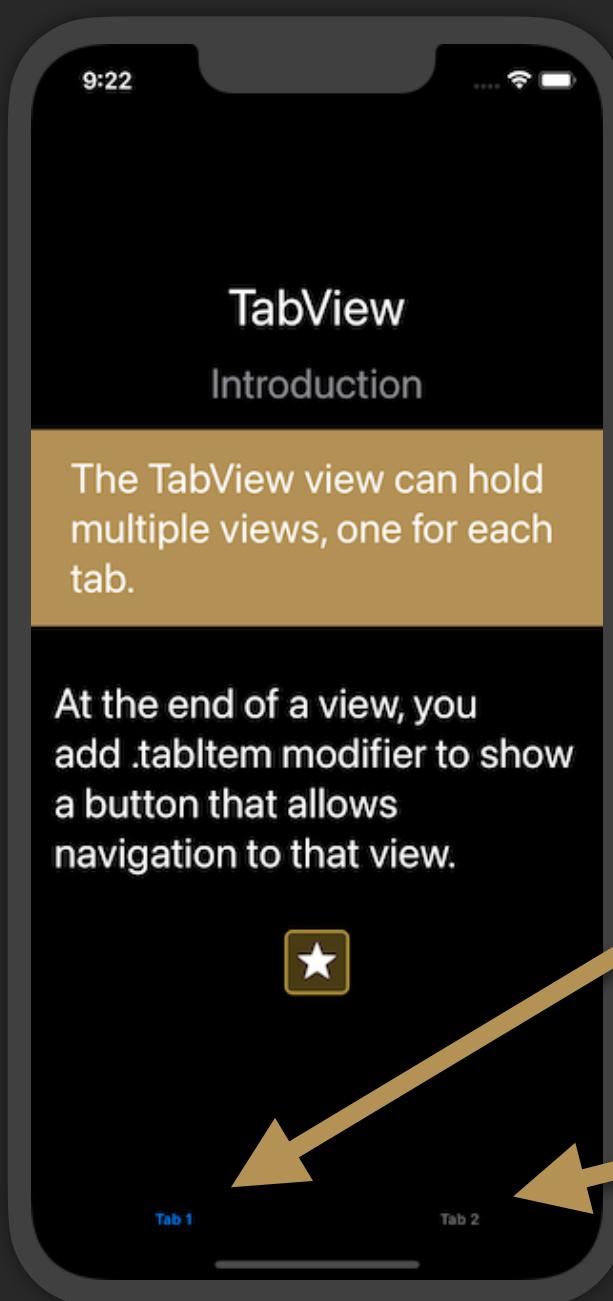


The TabView acts like a container for child views within it. These child views are individual screens. It provides tab buttons (TabItems) that allows the user to switch between these child views.

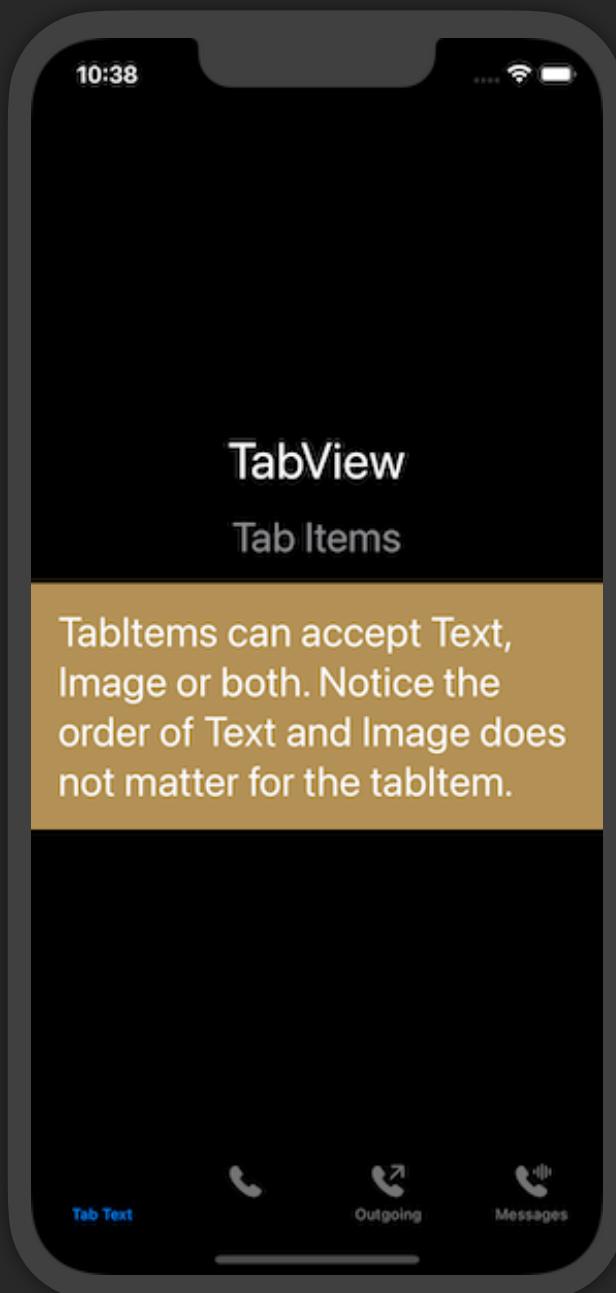
This is a push-out view.

Introduction

```
struct TabView_Intro : View {  
    var body: some View {  
        TabView {  
            // First Screen  
            VStack(spacing: 20) {  
                HeaderView("TabView",  
                    subtitle: "Introduction",  
                    desc: "The TabView view can hold multiple views, one for each tab.")  
  
                Text("At the end of a view, you add .tabItem modifier to show a button that  
                    allows navigation to that view.")  
                .padding()  
  
                Image("TabItem")  
  
            }  
            .tabItem {  
                // Creates a tab button in the tab bar  
                Text("Tab 1")  
            }  
  
            // Second Screen  
            Text("This view represents the Second Screen.")  
            .tabItem {  
                // Creates a tab button in the tab bar  
                Text("Tab 2")  
            }  
            .font(.title)  
        }  
    }  
}
```



TabItems



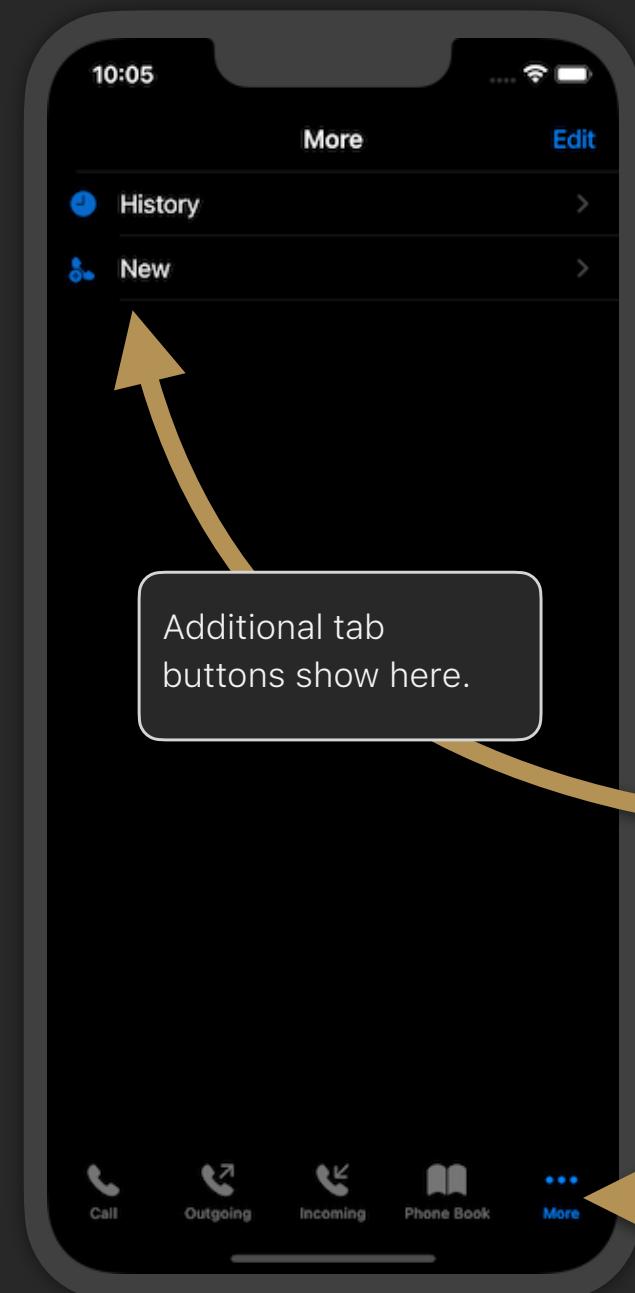
```
struct TabView_TabItems : View {
    var body: some View {
        TabView {
            TabOne()
            Text("Phone Calls")
            Text("Outgoing Phone Calls")
            Text("Messages")
        }
    }
}

struct TabOne: View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("TabView",
                subtitle: "Tab Items",
                desc: "TabItems can accept Text, Image or both. Notice the order of Text and Image does not matter for the tabItem.")
                .font(.title)
        }
    }
}
```

The diagram illustrates the structure of TabItems with the following annotations:

- Can be just text.** Points to the first tab item: `.tabItem { Text("Tab Text") }`.
- Or just an image.** Points to the second tab item: `.tabItem { Image(systemName: "phone") }`.
- Can use both. Order doesn't matter.** Points to the third tab item: `.tabItem { Image(systemName: "phone.arrow.up.right") } Text("Outgoing")` .
- iOS 14** Points to the fourth tab item: `.tabItem { Label("Messages", systemImage: "phone.and.waveform.fill") }`.
- You can even use a Label for the text and image.** Points to the fourth tab item: `.tabItem { Label("Messages", systemImage: "phone.and.waveform.fill") }`.

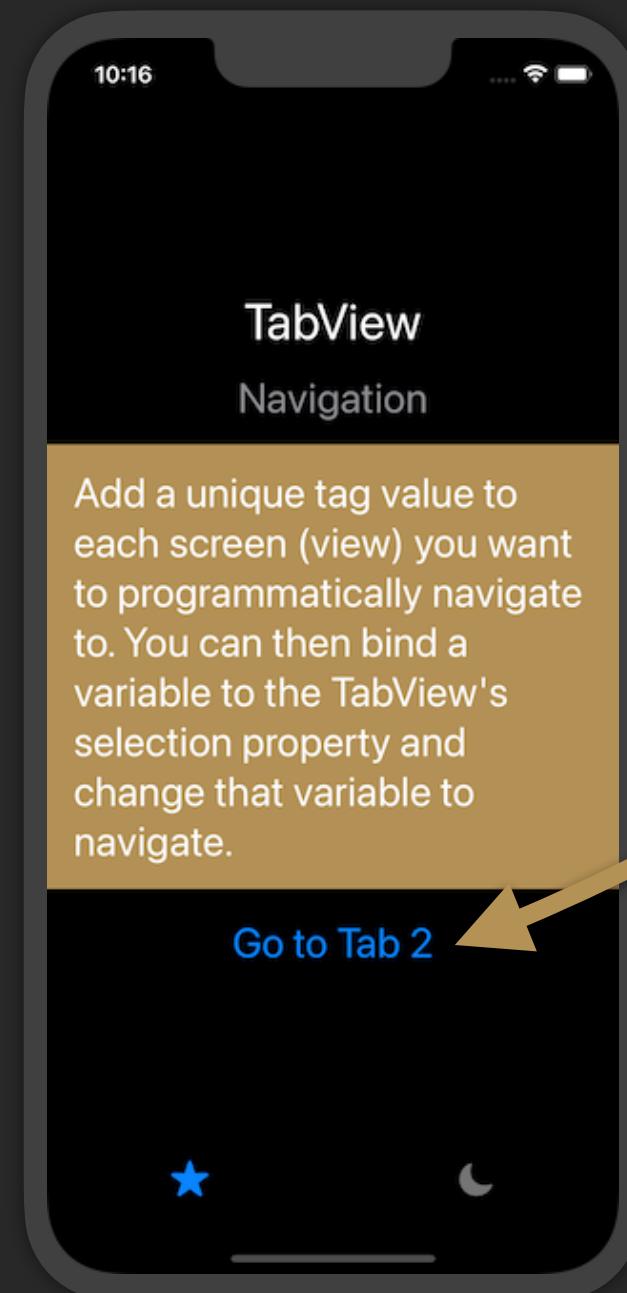
Too Many Tabs



```
TabView {  
    Text("Call Screen").tabItem {  
        Image(systemName: "phone")  
        Text("Call")  
    }  
    Text("Outgoing Phone Calls Screen").tabItem {  
        Image(systemName: "phone.arrow.up.right")  
        Text("Outgoing")  
    }  
    Text("Incoming Phone Calls Screen").tabItem {  
        Image(systemName: "phone.arrow.down.left")  
        Text("Incoming")  
    }  
    Text("Phone Book Screen").tabItem {  
        Image(systemName: "book")  
        Text("Phone Book")  
    }  
    Text("History Screen").tabItem {  
        Image(systemName: "clock")  
        Text("History")  
    }  
    Text("New Phone Number").tabItem {  
        Image(systemName: "phone.badge.plus")  
        Text("New")  
    }  
}
```

When there are too many tabs to fit for the device, the **More** button is created where you can find the rest of the tabs listed out.

Navigation



```
struct TabView_Navigating : View {  
    @State private var selectedTab = 1 // Set which tab is active  
  
    var body: some View {  
        // Tell the TabView which variable to listen to for changes  
        TabView(selection: $selectedTab) {  
            // Tab 1  
            VStack(spacing: 20) {  
                HeaderView("TabView",  
                    subtitle: "Navigation",  
                    desc: "Add a unique tag value to each screen (view) you want to  
programmatically navigate to. You can then bind a variable to the TabView's  
selection property and change that variable to  
navigate.")  
  
                Button("Go to Tab 2") {  
                    selectedTab = 2  
                }  
                .tabItem {  
                    Image(systemName: "star.fill")  
                }  
                .tag(1)  
            }  
            // Tab 2  
            VStack {  
                Text("Second Screen")  
            }  
            .tabItem {  
                Image(systemName: "moon.fill")  
            }  
            .tag(2)  
        }  
        .font(.title)  
    }  
}
```

Change the state property bound to the TabView's selection parameter to navigate to a different tab.

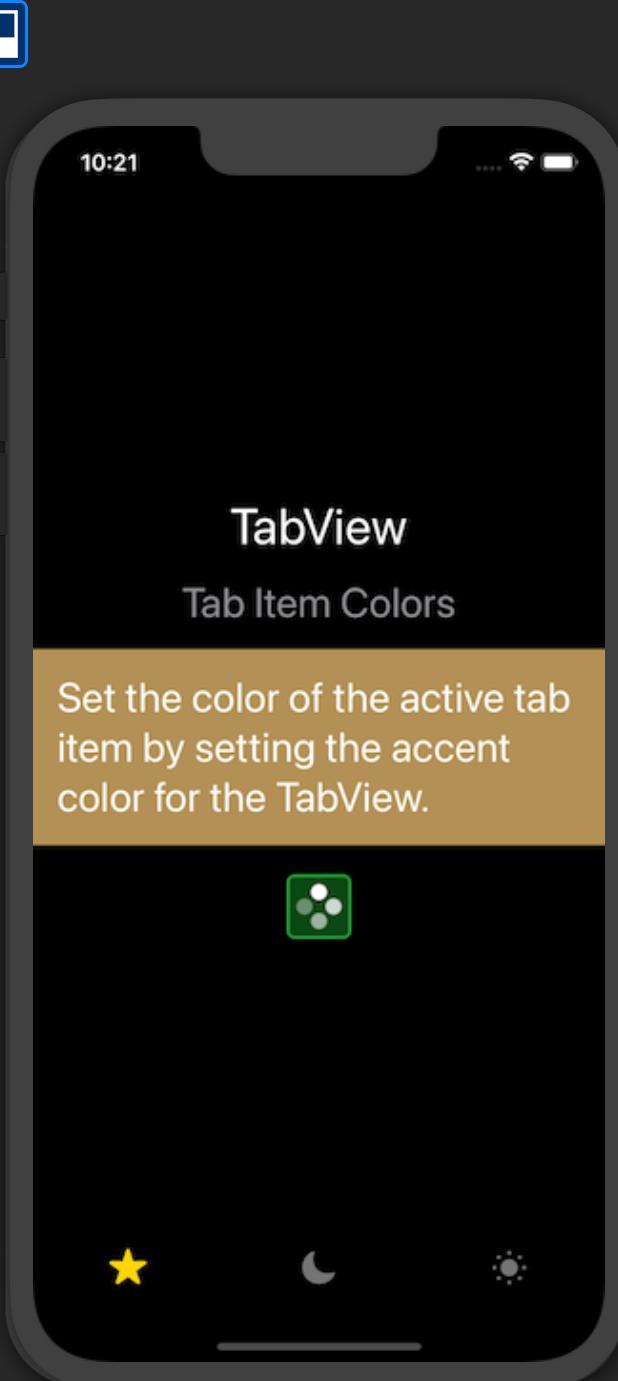
Add tags to enable programmatically navigating to tabs.

Colors

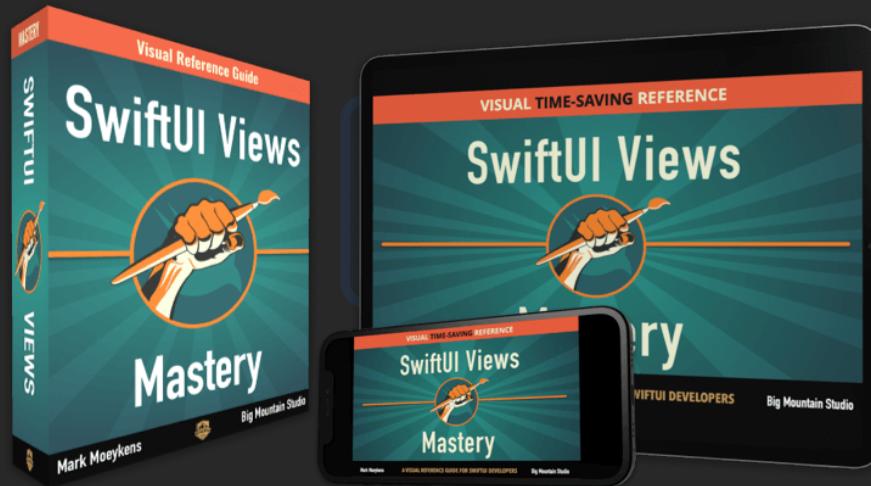
```
struct TabView_Colors : View {  
    var body: some View {  
        TabView {  
            // Tab 1  
            VStack(spacing: 20) {  
                HeaderView("TabView",  
                    subtitle: "Tab Item Colors",  
                    desc: "Set the color of the active tab item by setting the accent  
color for the TabView.")  
  
                Image("AccentColor")  
            }  
            .tabItem {  
                Image(systemName: "star.fill")  
            }  
            // Tab 2  
            Text("Second Screen")  
            .tabItem {  
                Image(systemName: "moon.fill")  
            }  
            .foregroundColor(Color.red)  
            // Tab 3  
            Text("Third Screen")  
            .tabItem {  
                Image(systemName: "sun.min.fill")  
            }  
        }  
        .font(.title)  
        .accentColor(.yellow)  
    }  
}
```

Notice that I am setting the foreground color of the second tabItem to red. This will have no effect on the color of the tab item. The background modifier will not work either.

The only thing that works is to set the accent color on the TabView itself.



TabView - Paging Style



This SwiftUI chapter is **locked** in this

With the existing TabView, this chapter now offers a new style for it that allows the views within a TabView to be able to be swiped horizontally and "snap" into place when the view enters the screen.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

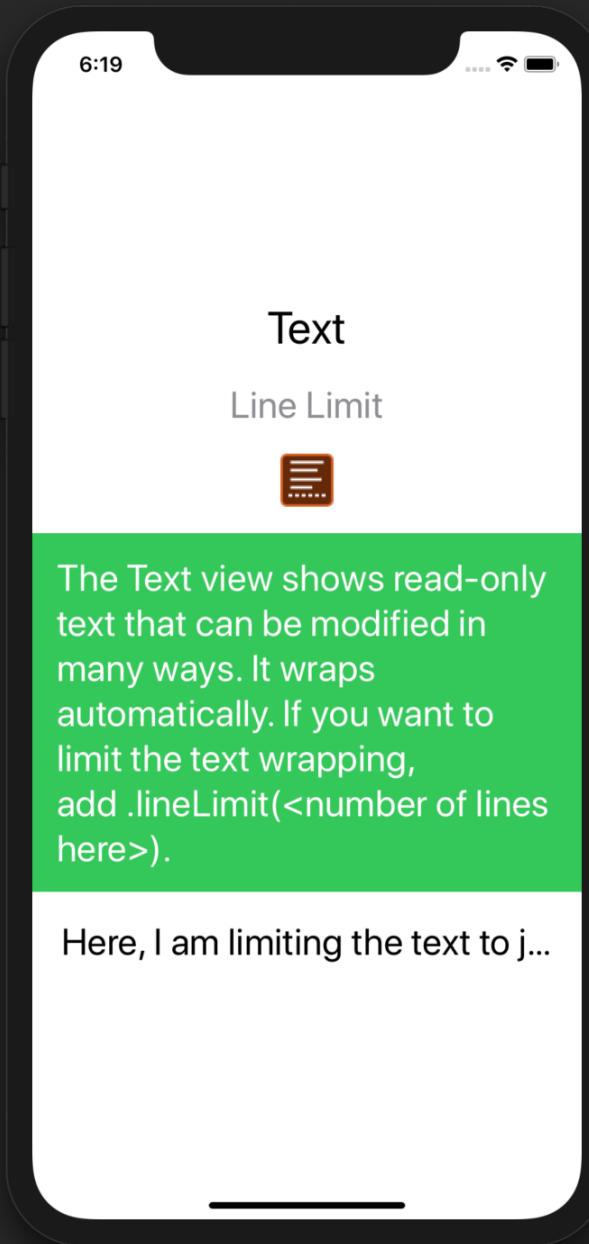
Text



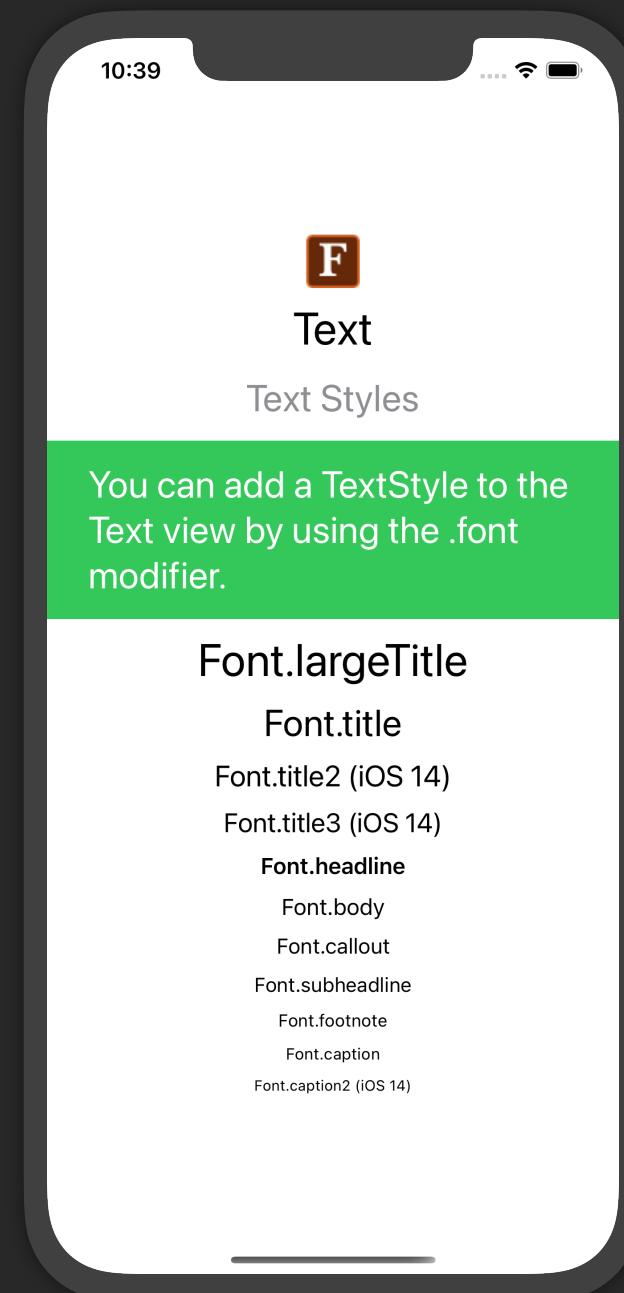
The text view will probably be one of your most-used views. It has many, if not the most, modifiers available to it.

This is a pull-in view.

Line Limit

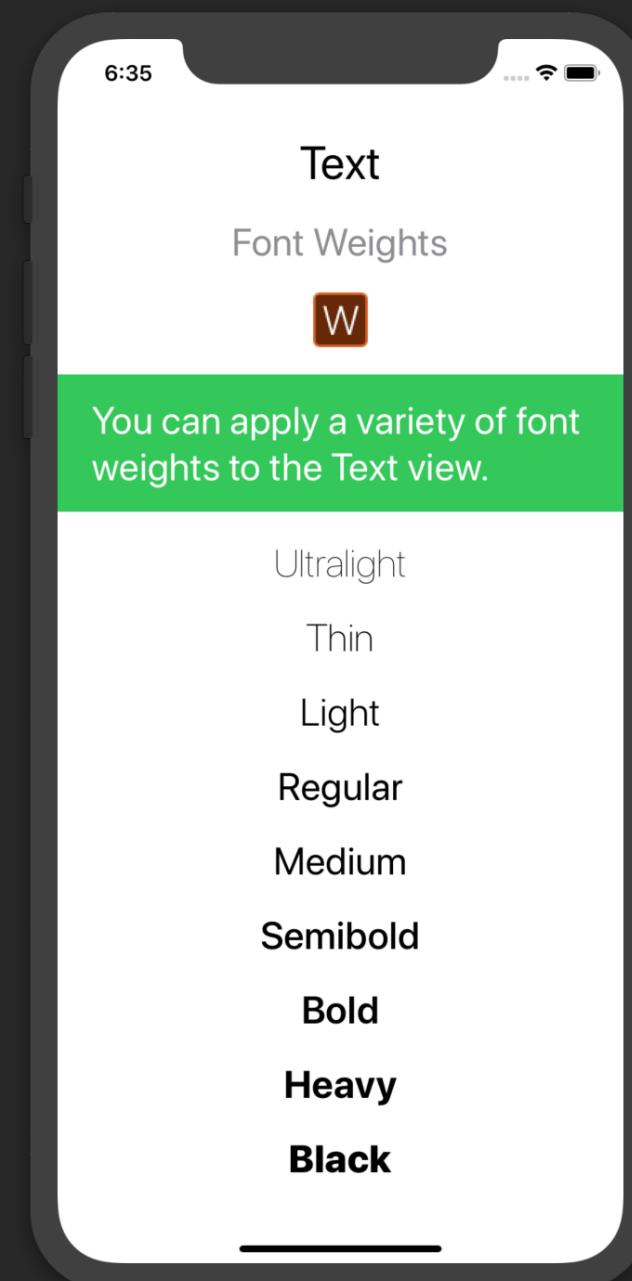


```
VStack(spacing: 20) {  
    Text("Text")  
        .font(.largeTitle)  
  
    Text("Line Limit")  
        .font(.title)  
        .foregroundColor(.gray)  
  
    Image("LineLimit")  
  
    Text("The Text view shows read-only text that can be modified in many ways. It wraps  
automatically. If you want to limit the text wrapping, add .lineLimit(<number of lines here>).")  
    ...  
  
    Text("Here, I am limiting the text to just one line.")  
        .lineLimit(1)  
        .font(.title)  
        .padding(.horizontal)  
}
```



Text Styles

```
struct Text_TextStyles: View {  
    var body: some View {  
        VStack(spacing: 10) {  
            Image("Font")  
  
            HeaderView("Text",  
                      subtitle: "Text Styles",  
                      desc: "You can add a TextStyle to the Text view by using the .font  
                             modifier.",  
                      back: .green, textColor: .white)  
                .font(.title)  
  
            Group {  
                Text("Font.largeTitle").font(.largeTitle)  
                Text("Font.title").font(.title)  
                Text("Font.title2 (iOS 14)").font(.title2) iOS 14  
                Text("Font.title3 (iOS 14)").font(.title3)  
            }  
            Group {  
                Text("Font.headline").font(.headline)  
                Text("Font.body").font(.body)  
                Text("Font.callout").font(.callout)  
                Text("Font.subheadline").font(.subheadline)  
                Text("Font.footnote").font(.footnote)  
                Text("Font.caption").font(.caption)  
                Text("Font.caption2 (iOS 14)").font(.caption2) iOS 14  
            }  
        }  
    }  
}
```



Weights

```
Text("Text")
    .font(.largeTitle)
Text("Font Weights")
    .font(.title)
    .foregroundColor(.gray)
Image("FontWeight")
Text("You can apply a variety of font weights to the Text view.")
    .padding()
    .frame(maxWidth: .infinity)
    .background(Color.green)
    .foregroundColor(.white)
    .font(.title)

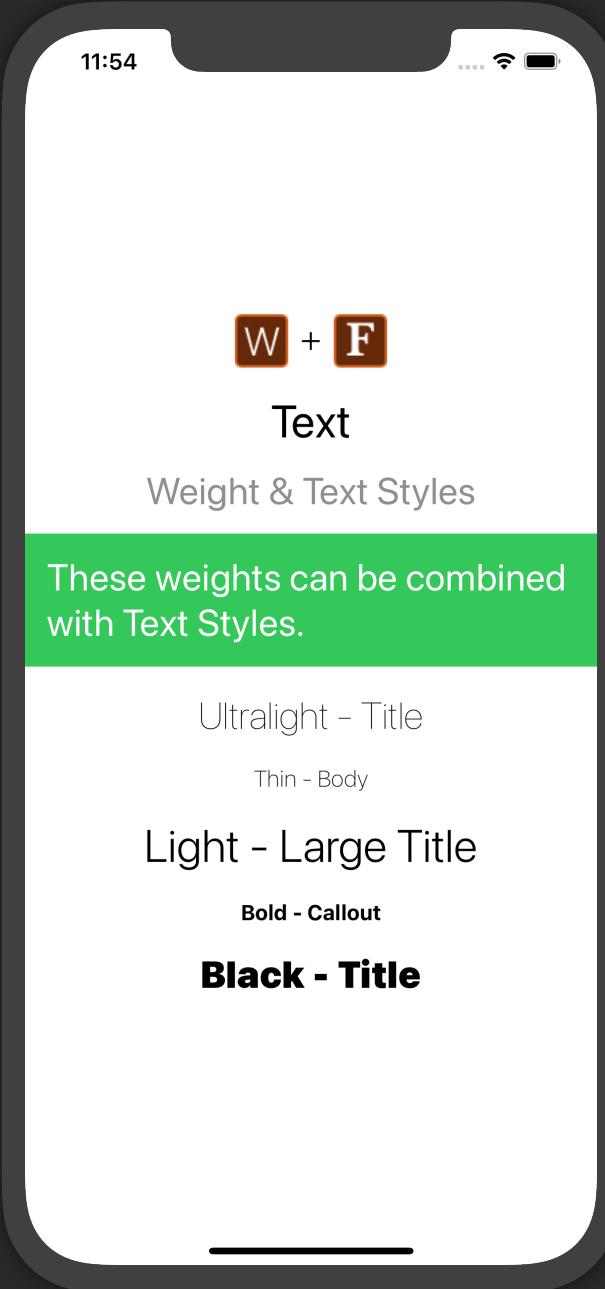
Group { // Too many views (> 10) in one container
    Text("Ultralight")
        .fontWeight(.ultraLight)
    Text("Thin")
        .fontWeight(.thin)
    Text("Light")
        .fontWeight(.light)
    Text("Regular")
        .fontWeight(.regular)
    Text("Medium")
        .fontWeight(.medium)
    Text("Semibold")
        .fontWeight(.semibold)
    Text("Bold")
        .fontWeight(.bold)
    Text("Heavy")
        .fontWeight(.heavy)
    Text("Black")
        .fontWeight(.black)
}.font(.title)
```

Note: The `fontWeight` modifier can ONLY be applied to Text views.

Unlike the `font` modifier which can be applied to any view.

To apply weight to any view using the `font` modifier, see next page.

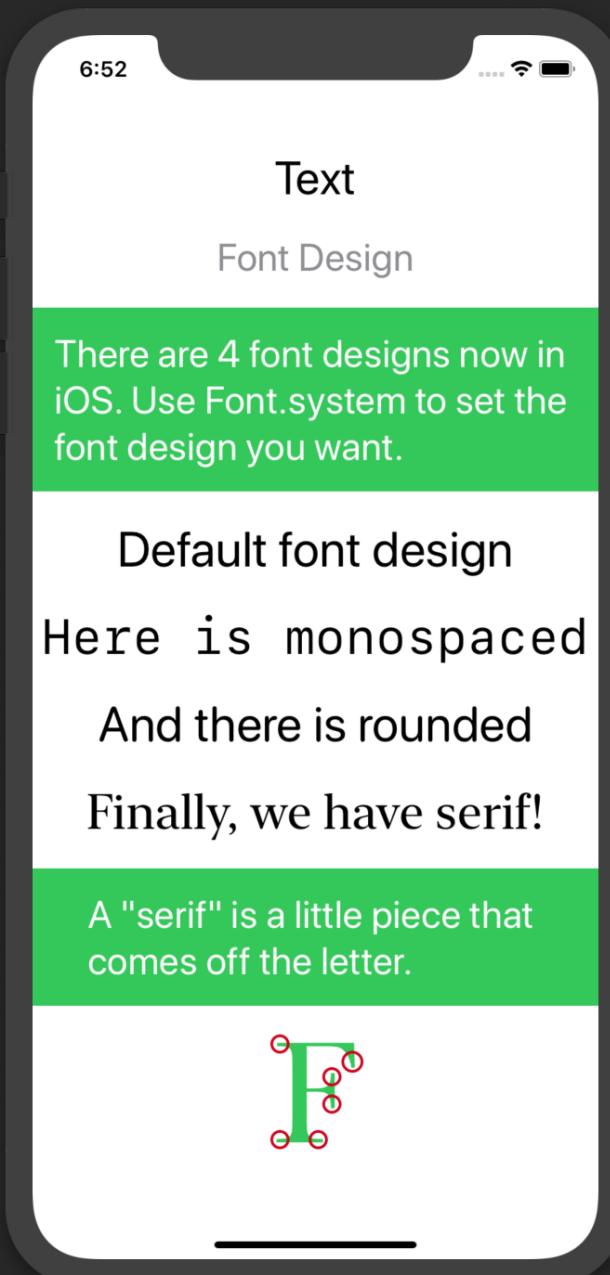
Weight & Text Style Combined



```
struct Text_Weights_TextStyles: View {  
    var body: some View {  
        return VStack(spacing: 20) {  
            HStack {  
                Image("FontWeight")  
                Image(systemName: "plus")  
                Image("Font")  
            }  
  
            HeaderView("Text", subtitle: "Weight & Text Styles",  
                      desc: "These weights can be combined with Text Styles.",  
                      back: .green, textColor: .white)  
                .font(.title)  
  
            Text("Ultralight - Title")  
                .fontWeight(.ultraLight)  
                .font(.title)  
            Text("Thin - Body")  
                .fontWeight(.thin)  
                .font(.body)  
            Text("Light - Large Title")  
                .fontWeight(.light)  
                .font(.largeTitle)  
            Text("Bold - Callout")  
                .fontWeight(.bold)  
                .font(.callout)  
            Text("Black - Title")  
                .font(Font.title.weight(.black))  
        }  
    }  
}
```

Instead of two modifiers, you can combine text style and weight in just ONE modifier like this.

Font Design



```
struct Text_FontDesign : View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("Text", subtitle: "Font Design", desc: "There are 4 font designs now in iOS. Use Font.system to set the font design you want.", back: .green, textColor: .white)

            Text("Default font design")
                .font(Font.system(size: 30, design: Font.Design.default))

            // You can remove the "Font.Design" of the enum
            Text("Here is monospaced")
                .font(.system(size: 30, design: .monospaced))

            Text("And there is rounded")
                .font(.system(.title, design: .rounded))

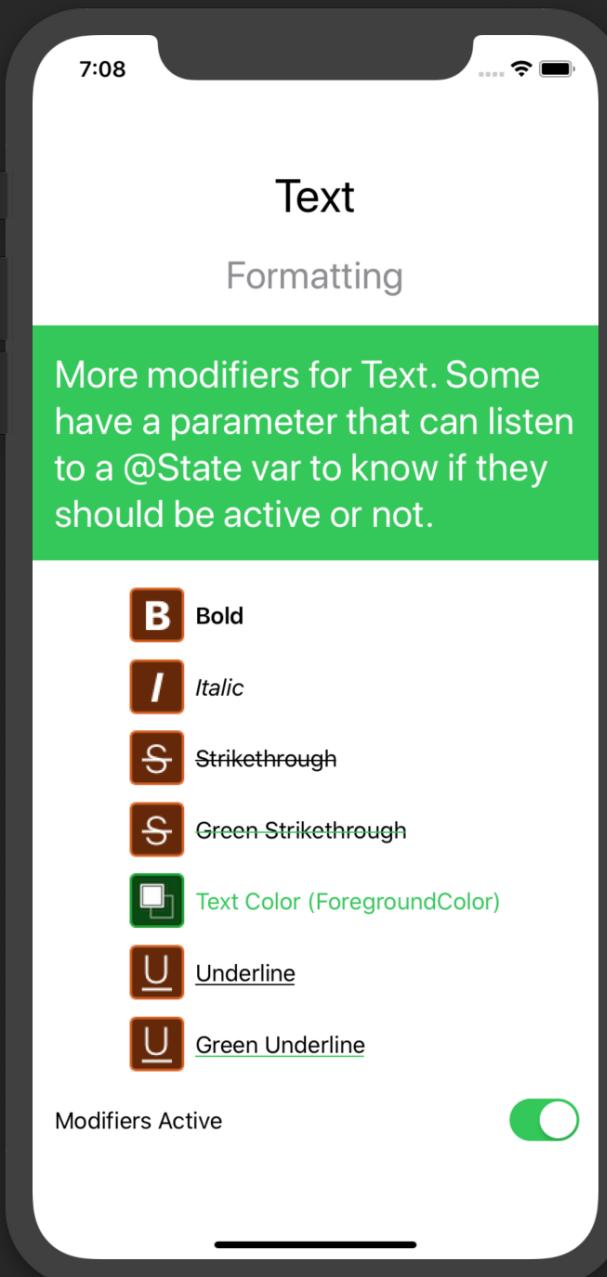
            Text("Finally, we have serif!")
                .font(.system(.title, design: .serif))

            DescView(desc: "A \"serif\" is a little piece that comes off the letter.", back: .green, textColor: .white)

            Image("Serif")
        }
        .font(.title)
    }
}
```

Set the design with a hard-coded size or use a text style.

Formatting

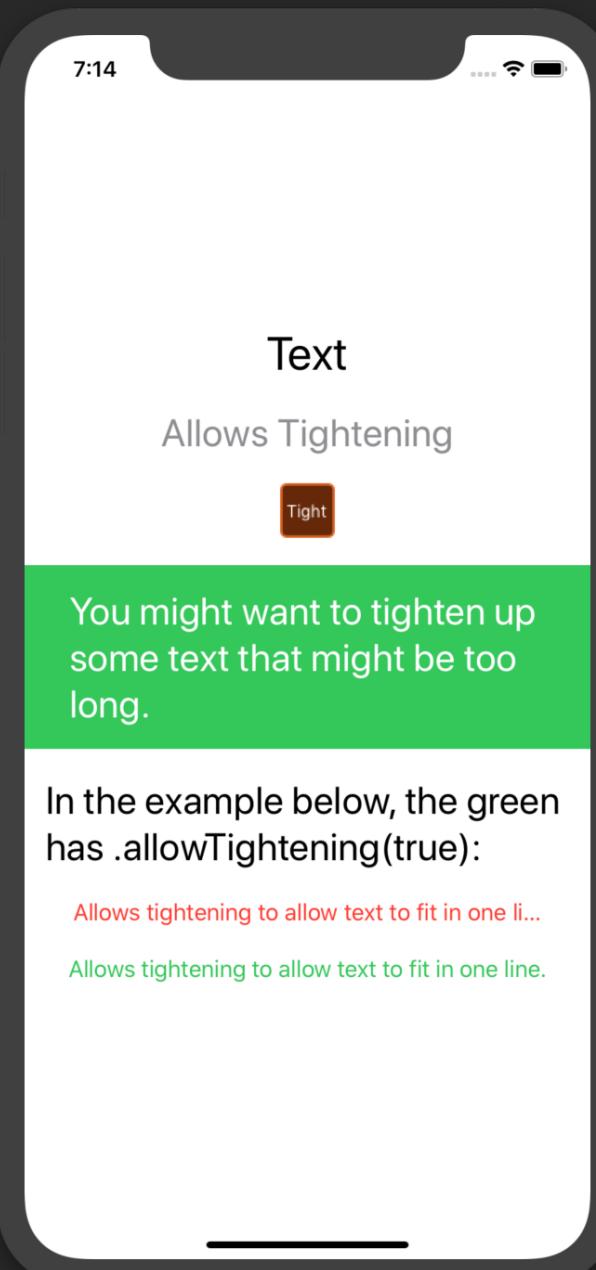


```

@State private var modifierActive = true
...
HStack {
    Image("Bold")
    Text("Bold").bold()
}
HStack {
    Image("Italic")
    Text("Italic").italic()
}
HStack {
    Image("Strikethrough")
    Text("Strikethrough").strikethrough()
}
HStack {
    Image("Strikethrough")
    Text("Green Strikethrough")
        .strikethrough(modifierActive, color: .green)
}
HStack {
    Image("ForegroundColor")
    Text("Text Color (ForegroundColor)").foregroundColor(.green)
}
HStack {
    Image("Underline")
    Text("Underline").underline()
}
HStack {
    Image("Underline")
    Text("Green Underline").underline(modifierActive, color: .green)
}
...
Toggle("Modifiers Active", isOn: $modifierActive)

```

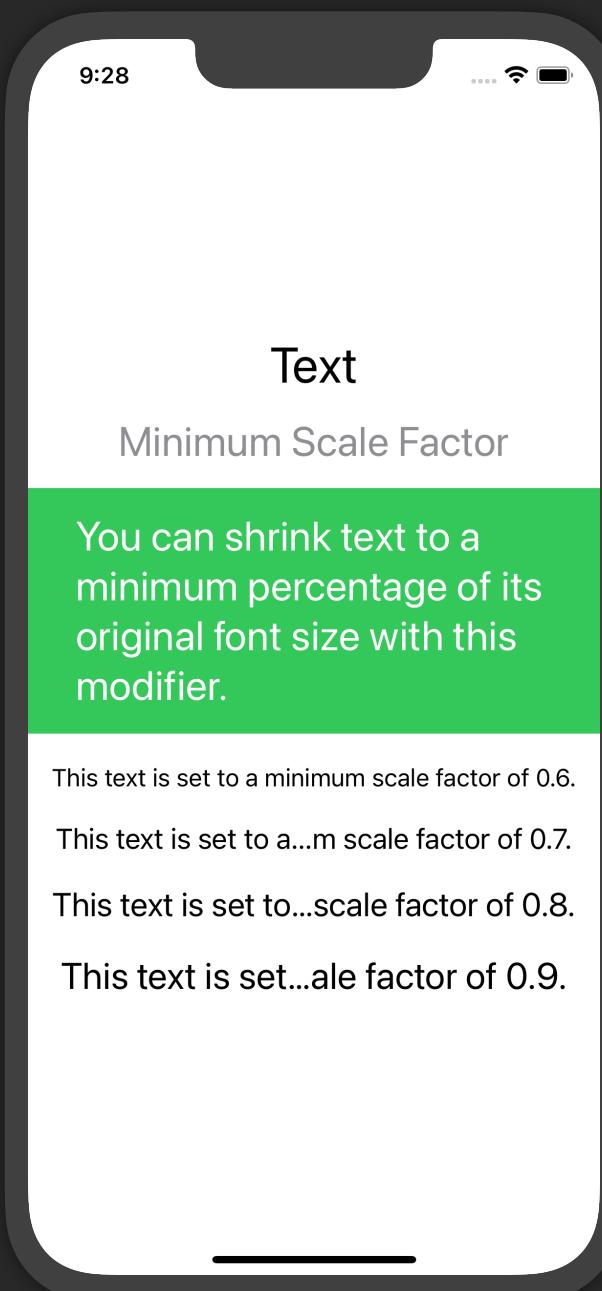
Allows Tightening



```
VStack(spacing: 20) {  
    ...  
  
    Image("AllowsTightening")  
  
    Text("You might want to tighten up some text that might be too long.")  
    ...  
  
    Text("In the example below, the green has .allowTightening(true)")  
    ...  
  
    Group {  
        Text("Allows tightening to allow text to fit in one line.")  
        .foregroundColor(.red)  
        .allowsTightening(false)  
        .padding(.horizontal)  
        .lineLimit(1)  
        Text("Allows tightening to allow text to fit in one line.")  
        .foregroundColor(.green)  
        .allowsTightening(true)  
        .padding(.horizontal)  
        .lineLimit(1)  
    }.padding(.horizontal)  
}
```

Allows Tightening can be helpful when you see the last word getting truncated. Applying it may not even fully work depending on just how much space can be tightened. With the default font, I notice I can get a couple of characters worth of space to tighten up.

Minimum Scale Factor

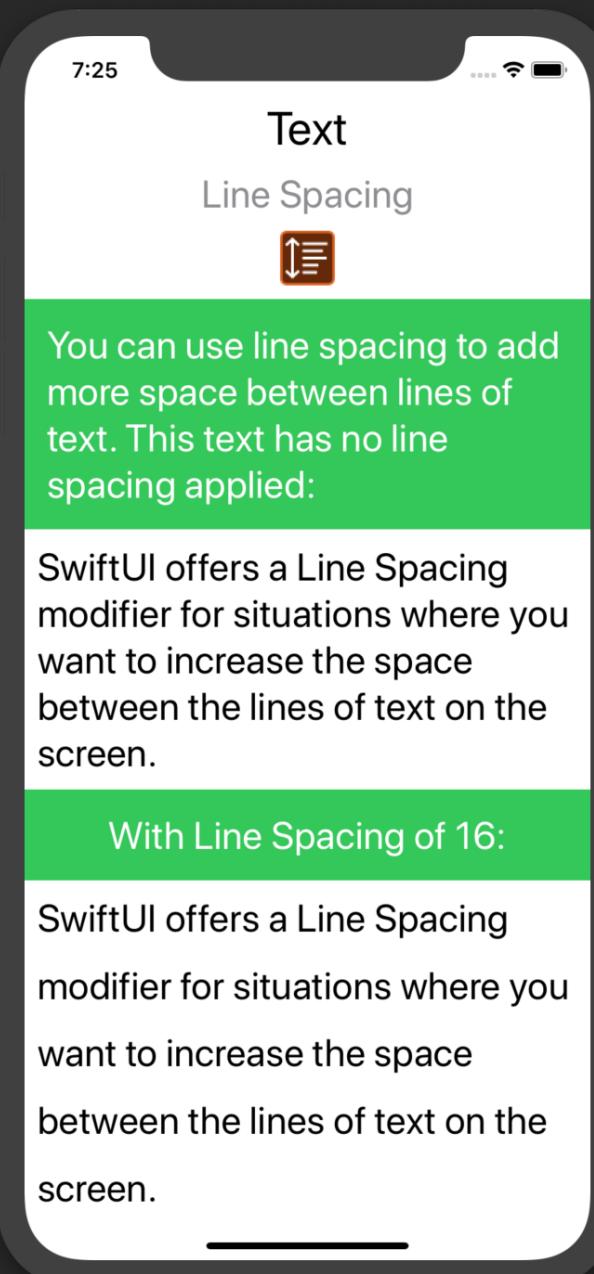


```
struct Text_MinimumScaleFactor : View {
    var body: some View {
        VStack(spacing: 20) {
            HeaderView("Text",
                subtitle: "Minimum Scale Factor",
                desc: "You can shrink text to a minimum percentage of its original font
size with this modifier.",
                back: .green, textColor: .white)

            Group {
                Text("This text is set to a minimum scale factor of 0.6.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.6)
                Text("This text is set to a minimum scale factor of 0.7.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.7)
                Text("This text is set to a minimum scale factor of 0.8.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.8)
                Text("This text is set to a minimum scale factor of 0.9.")
                    .lineLimit(1)
                    .minimumScaleFactor(0.9)
            }
            .truncationMode(.middle)
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```

.minimumScaleFactor takes a fraction from 0 to 1. For example, 0.3 is 30% of the original size of the font that it can shrink. If the font size is 100, then it can shrink to 30.

Line Spacing



```
 VStack(spacing: 10) {  
     Text("Text").font(.largeTitle)  
     Text("Line Spacing").font(.title).foregroundColor(.gray)  
     Image("LineSpacing")  
 }
```

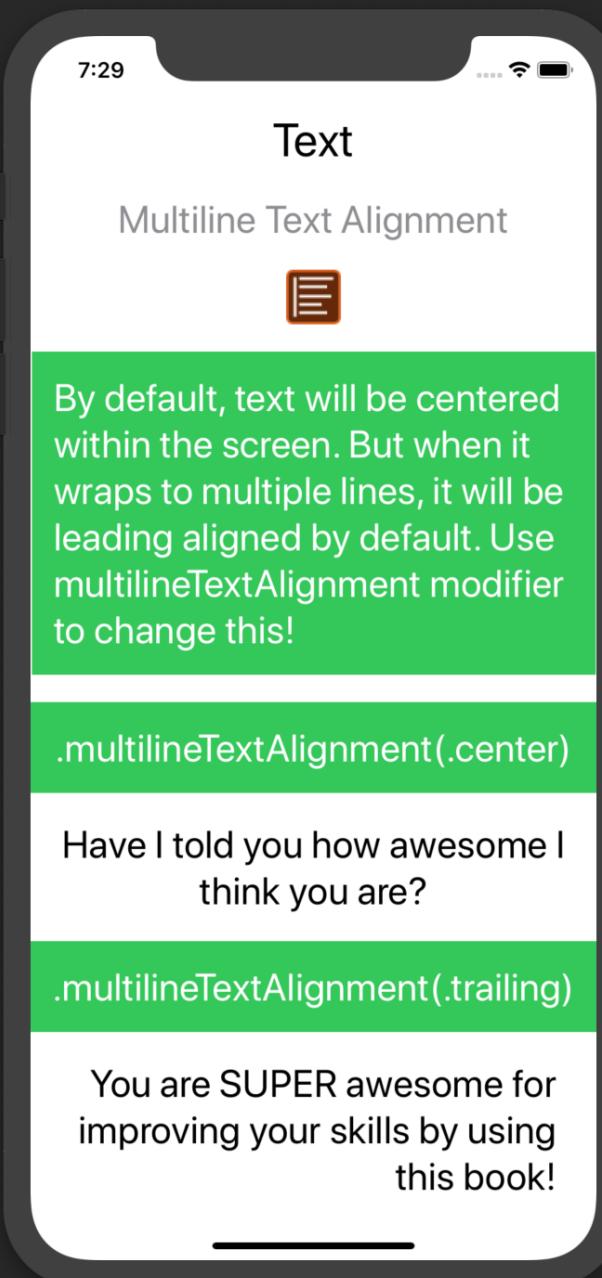
```
 Text("You can use line spacing to add more space between lines of text. This text has no  
 line spacing applied:")  
     .font(.title)  
     .frame(maxWidth: .infinity)  
     .padding()  
     .background(Color.green)  
     .foregroundColor(Color.white)
```

```
 Text("SwiftUI offers a Line Spacing modifier for situations where you want to increase the  
 space between the lines of text on the screen.")  
     .font(.title)
```

```
 Text("With Line Spacing of 16:")  
     .font(.title)  
     .frame(maxWidth: .infinity)  
     .padding()  
     .background(Color.green)  
     .foregroundColor(Color.white)
```

```
 Text("SwiftUI offers a Line Spacing modifier for situations where you want to increase the  
 space between the lines of text on the screen.")  
     .lineSpacing(16.0) // Add spacing between lines  
     .font(.title)  
 }
```

Alignment



```
 VStack(spacing: 20) {
    Text("Text").font(.largeTitle)
    Text("Multiline Text Alignment").foregroundColor(.gray)
    Image("MultilineTextAlignment")
    Text("By default, text will be centered within the screen. But when it wraps to multiple lines, it will be leading aligned by default. Use multilineTextAlignment modifier to change this!")
    ...
    Text(".multilineTextAlignment(.center)")
        .frame(maxWidth: .infinity)
        .padding()
        .foregroundColor(.white)
        .background(Color.green)

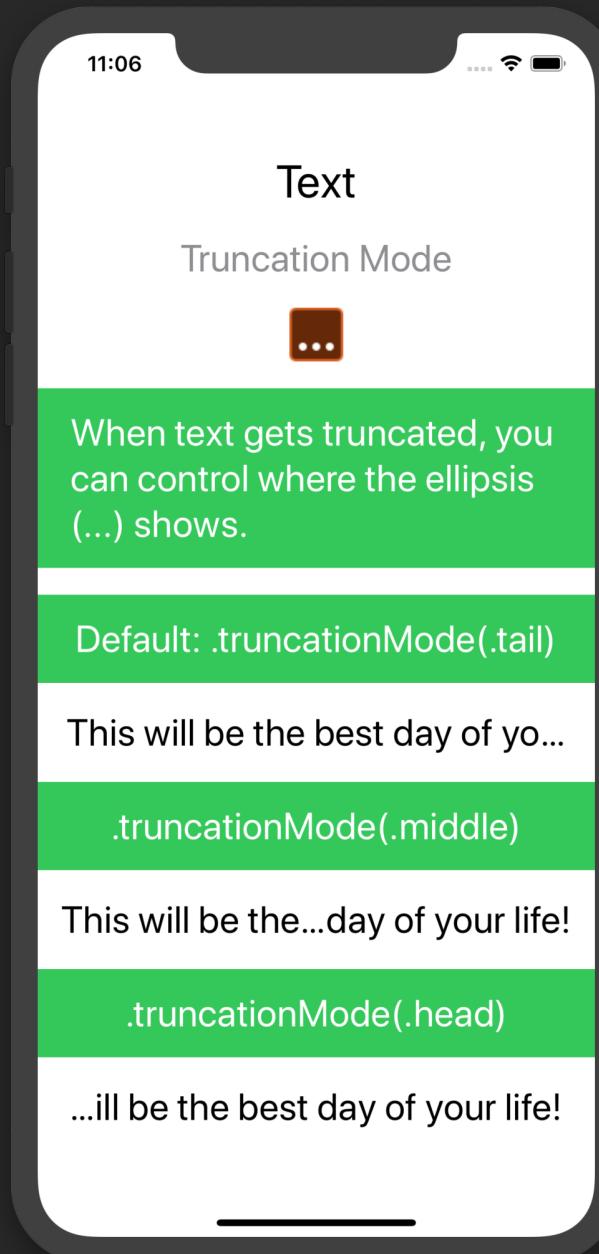
    Text("Have I told you how awesome I think you are?")
        .multilineTextAlignment(.center) // Center align
        .padding(.horizontal)

    Text(".multilineTextAlignment(.trailing)")
        .frame(maxWidth: .infinity)
        .padding()
        .foregroundColor(.white)
        .background(Color.green)
        .allowsTightening(true) // Prevent truncation

    Text("You are SUPER awesome for improving your skills by using this book!")
        .multilineTextAlignment(.trailing) // Trailing align
        .padding(.horizontal)
}
```

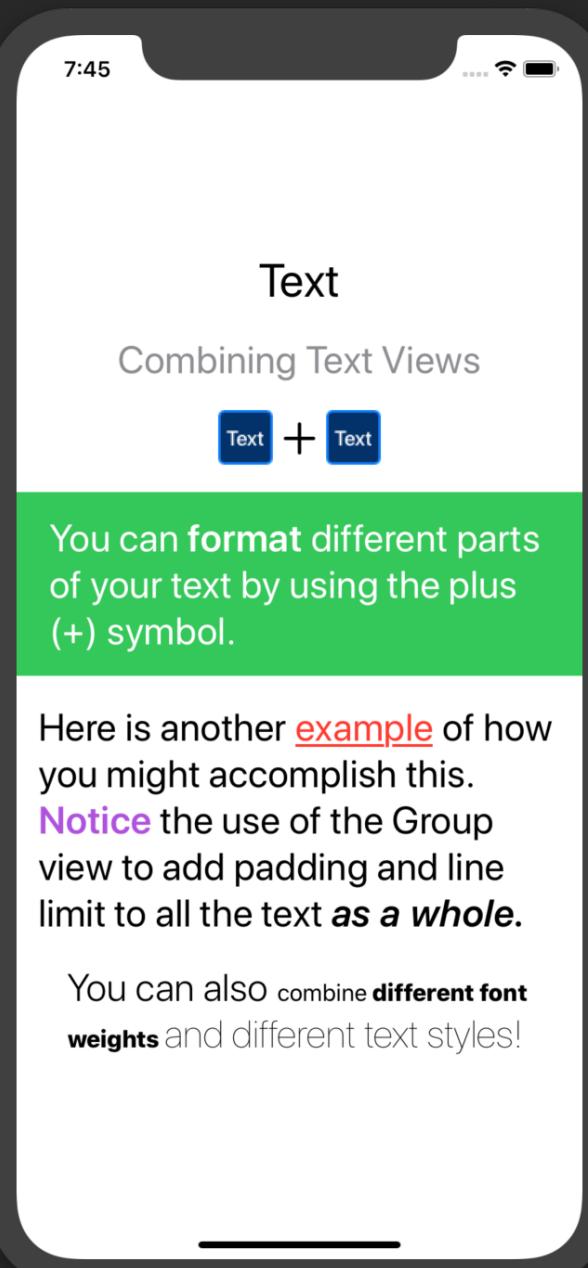
.font(.title) // Apply this text style to all text views

Truncation Mode



```
 VStack(spacing: 20) {
    Text("Text").font(.largeTitle)
    Text("Truncation Mode").font(.title).foregroundColor(.gray)
    Image("TruncationMode")
    Text("When text gets truncated, you can control where the ellipsis (...) shows.")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
        .font(.title)
    Text("Default: .truncationMode(.tail)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
        .font(.title)
    // Text automatically defaults at end
    Text("This will be the best day of your life!")
        .padding(.horizontal)
        .lineLimit(1)
    Text(".truncationMode(.middle)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
    Text("This will be the best day of your life!")
        .truncationMode(.middle) // Truncate in middle
        .padding(.horizontal)
        .lineLimit(1)
    Text(".truncationMode(.head)")
        .frame(maxWidth: .infinity).padding()
        .foregroundColor(.white).background(Color.green)
    Text("This will be the best day of your life!")
        .truncationMode(.head) // Truncate at beginning
        .padding(.horizontal)
        .lineLimit(1)
}
```

Combining Modified Text



```

Group {
    Text("You can ")
        + Text("format").bold()
        + Text (" different parts of your text by using the plus (+) symbol.")
}
...
Group {
    Text("Here is another ")
        + Text("example").foregroundColor(.red).underline()
        + Text (" of how you might accomplish this. ")
        + Text("Notice").foregroundColor(.purple).bold()
        + Text (" the use of the Group view to add padding and line limit to all the text ")
        + Text("as a whole.").bold().italic()
}
    .font(.title)
    .padding(.horizontal)

```

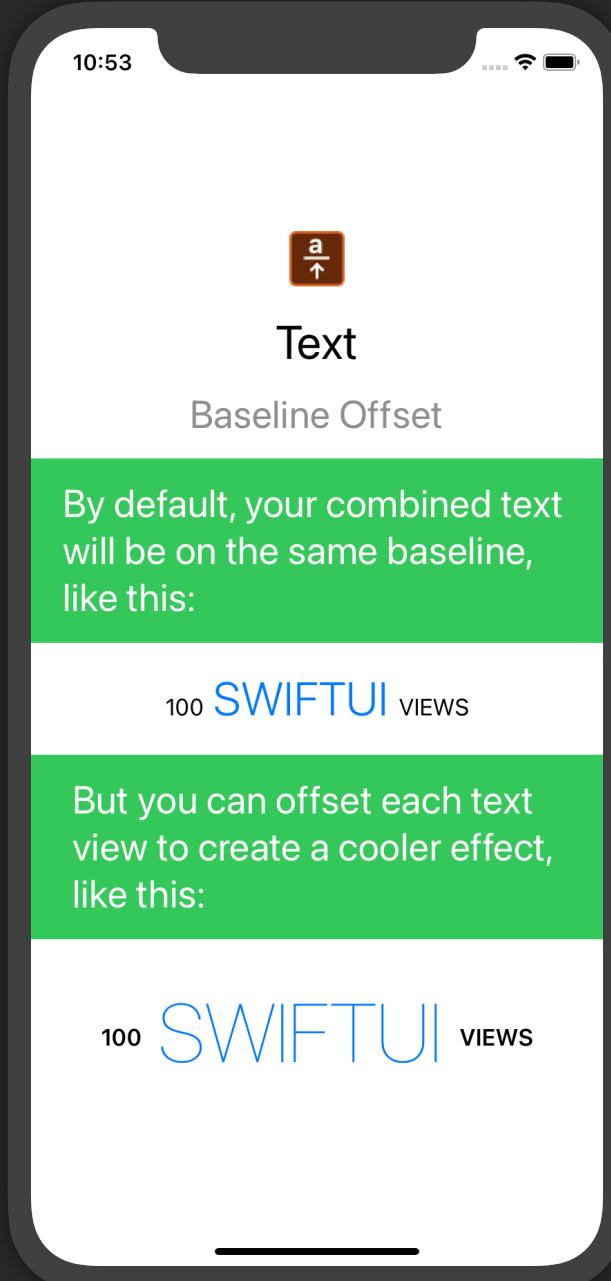
```

Group {
    Text("You can also ").font(.title).fontWeight(.light)
        + Text("combine")
        + Text(" different font weights ").fontWeight(.black)
        + Text("and different text styles!").font(.title).fontWeight(.ultraLight)
}
    .padding(.horizontal)

```

Although you see I'm wrapping my Text views in a Group, it is not required. I only do this so I can apply common modifiers to everything within the Group. See section on the Group view for more information.

Baseline Offset



```

struct Text_BaselineOffset : View {
    var body: some View {
        VStack(spacing: 20) {
            Image("BaselineOffset")
            HeaderView("Text",
                subtitle: "Baseline Offset",
                desc: "By default, your combined text will be on the same baseline, like
this:", back: .green, textColor: .white)
                .font(.title)
                .layoutPriority(1)

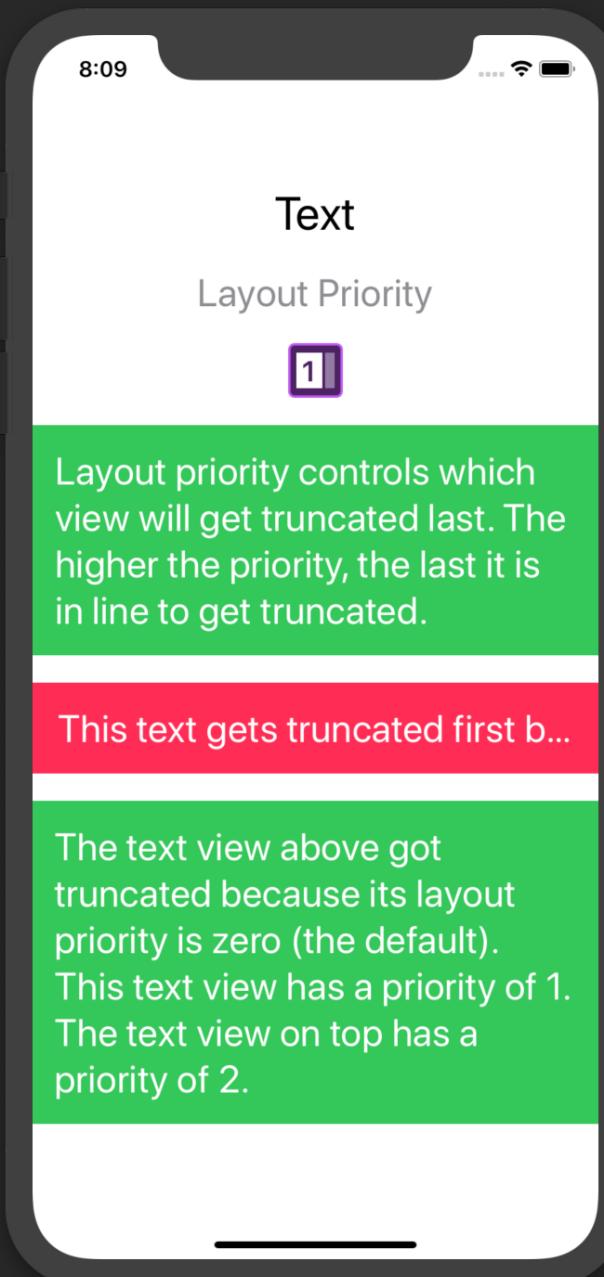
            Text("100")
                + Text(" SWIFTUI ").font(.largeTitle).fontWeight(.light)
                .foregroundColor(.blue)
                + Text ("VIEWS")

            DescView(desc: "But you can offset each text view to create a cooler effect, like
this:", back: .green, textColor: .white)
                .font(.title)

            Group {
                Text("100").bold()
                    + Text(" SWIFTUI ")
                    .font(Font.system(size: 60))
                    .fontWeight(.ultraLight)
                    .foregroundColor(.blue)
                    .baselineOffset(-12) // Negative numbers make it go down
                    + Text ("VIEWS").bold()
            }
        }
    }
}

```

Layout Priority



```
Text("Text")
    .font(.largeTitle)
Text("Layout Priority")
    .font(.title)
    .foregroundColor(.gray)

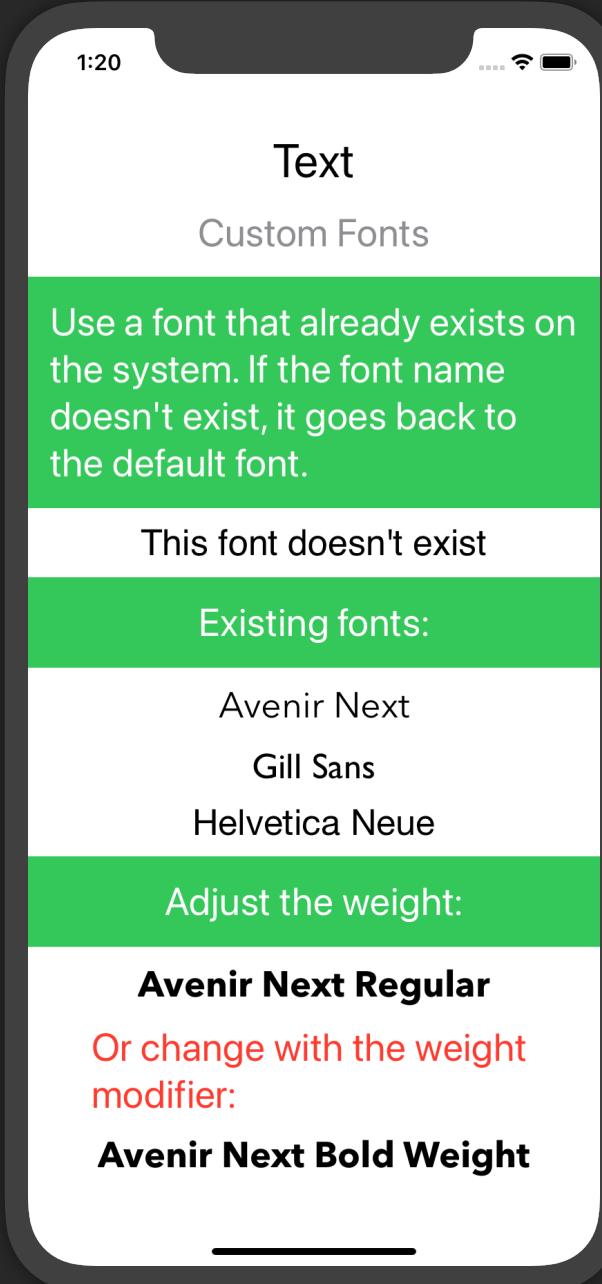
Image("LayoutPriority")

Text("Layout priority controls which view will get truncated last. The higher the priority, the
last it is in line to get truncated.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.green)
    .layoutPriority(2) // Highest priority to get the space it needs

Text("This text gets truncated first because it has no priority.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.pink)

Text("The text view above got truncated because its layout priority is zero (the default). This
text view has a priority of 1. The text view on top has a priority of 2.")
    .font(.title)
    .foregroundColor(.white)
    .frame(maxWidth: .infinity)
    .padding()
    .background(Color.green)
    .layoutPriority(1) // Next highest priority
```

Custom Fonts



```

struct Text_CustomFont: View {
    var body: some View {
        VStack(spacing: 10) {
            HeaderView("Text",
                subtitle: "Custom Fonts",
                desc: "Use a font that already exists on the system. If the font name
does't exist, it goes back to the default font.", back: .green, textColor: .white)

            Text("This font doesn't exist")
                .font(Font.custom("No Such Font", size: 26))

            DescView(desc: "Existing fonts:", back: .green, textColor: .white)

            Text("Avenir Next")
                .font(Font.custom("Avenir Next", size: 26))

            Text("Gill Sans")
                .font(Font.custom("Gill Sans", size: 26))

            Text("Helvetica Neue")
                .font(Font.custom("Helvetica Neue", size: 26))

            DescView(desc: "Adjust the weight:", back: .green, textColor: .white)

            Text("Avenir Next Regular")
                .font(Font.custom("Avenir Next Bold", size: 26))

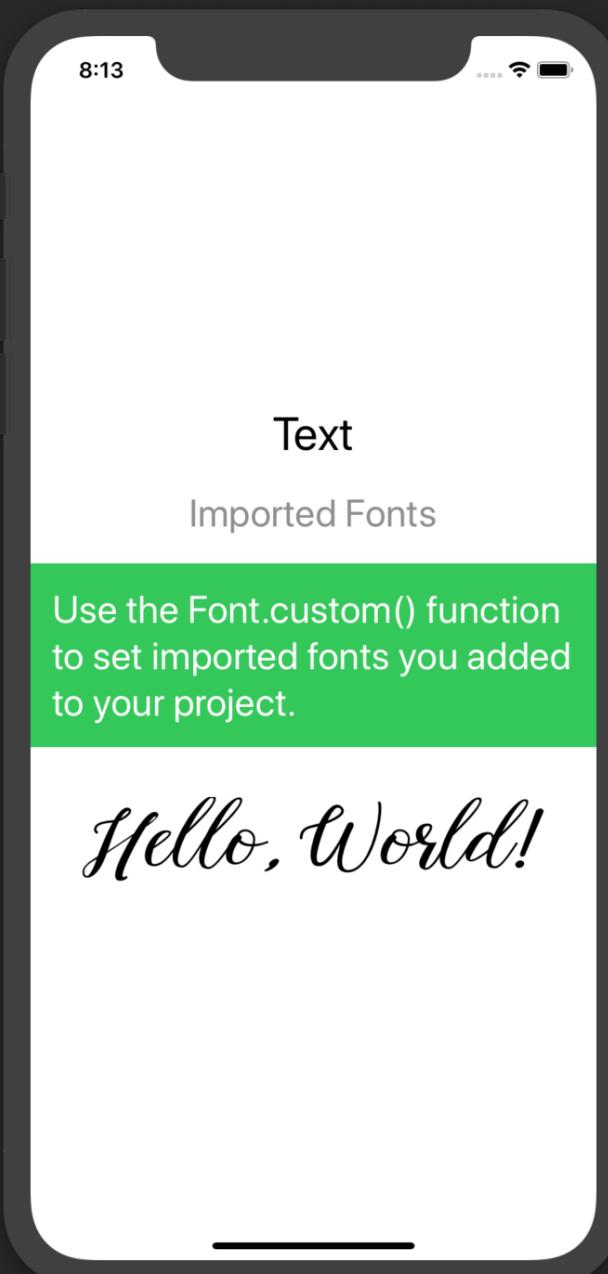
            Text("Or change with the weight modifier:")
                .foregroundColor(.red)

            Text("Avenir Next Bold Weight")
                .font(Font.custom("Avenir Next", size: 26).weight(.bold))
        }
        .font(.title)
        .ignoresSafeArea(edges: .bottom)
    }
}

```

Imported Fonts

```
struct Text_CustomFont: View {  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("Text")  
                .font(.largeTitle)  
  
            Text("Imported Fonts")  
                .font(.title)  
                .foregroundColor(.gray)  
  
            Text("Use the Font.custom() function to set imported fonts you added to your  
project.")  
            ...  
  
            Text("Hello, World!")  
                .font(Font.custom("Nightcall", size: 60))  
                .padding(.top)  
        }  
    }  
}
```



In order for this to work, you have to add the font file to your project and be sure to have the font file target your project. Then you need to add the font file name to the Info.plist under the "Fonts provided by application" key:

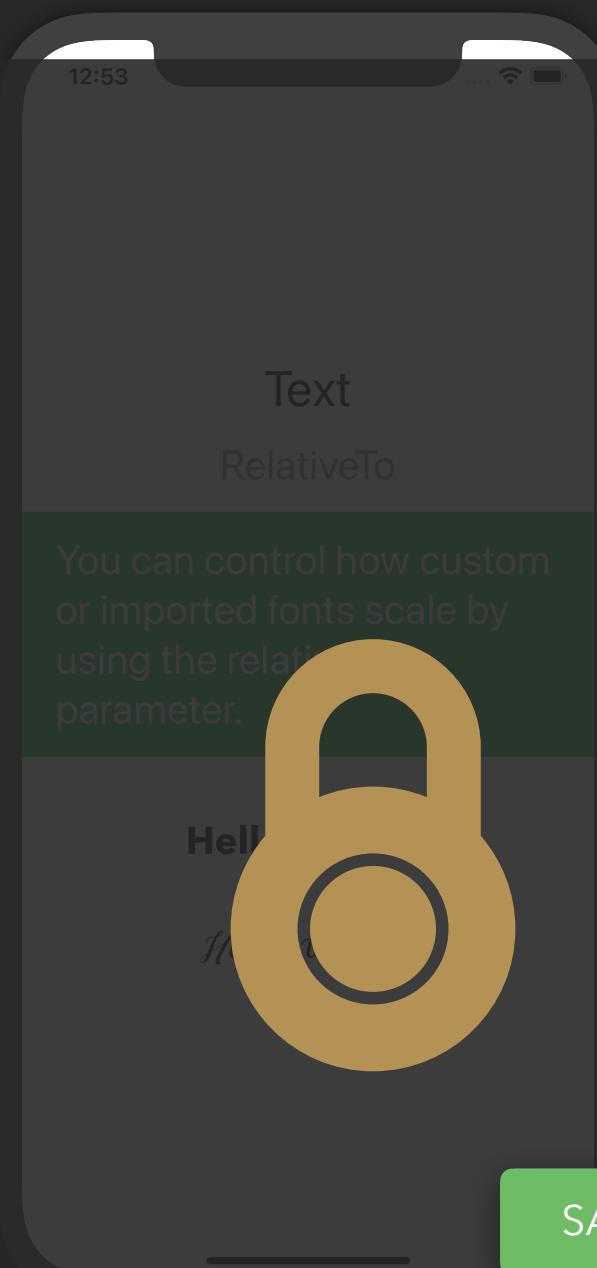
▶ Supported interface orientations (i... ▾	Array	(4 items)
▼ Fonts provided by application	Array	(1 item)
Item 0	String	Nightcall.ttf

Custom Font Size & `RelativeTo`

iOS 14

iOS 15

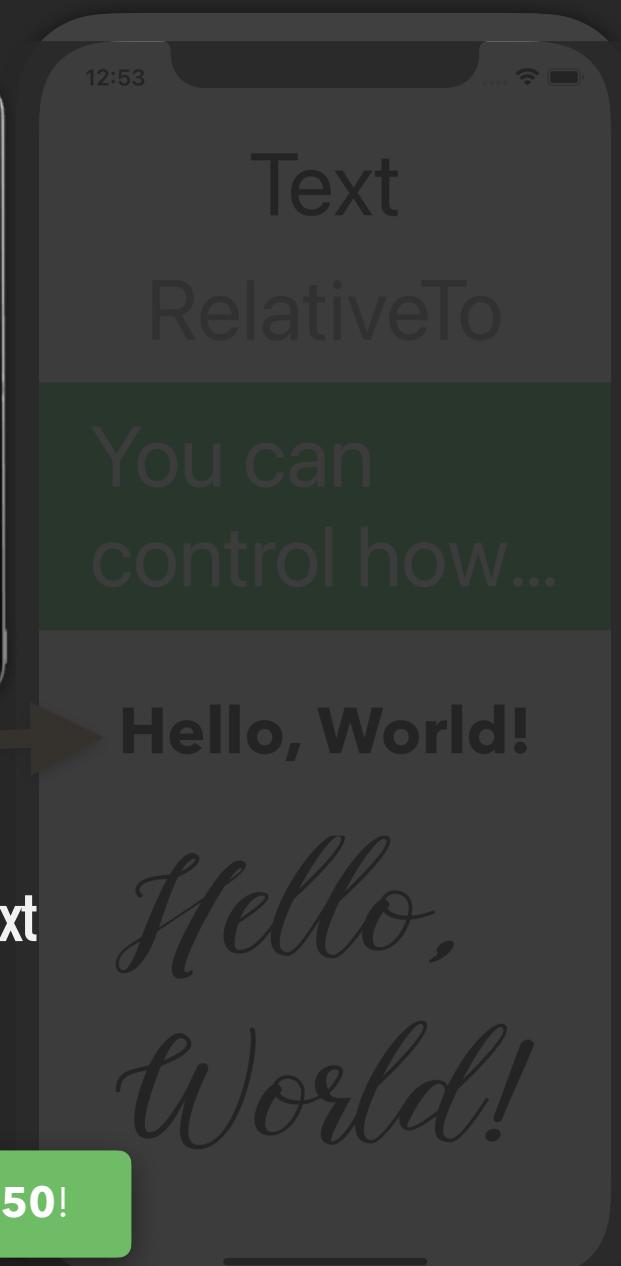
iOS 16



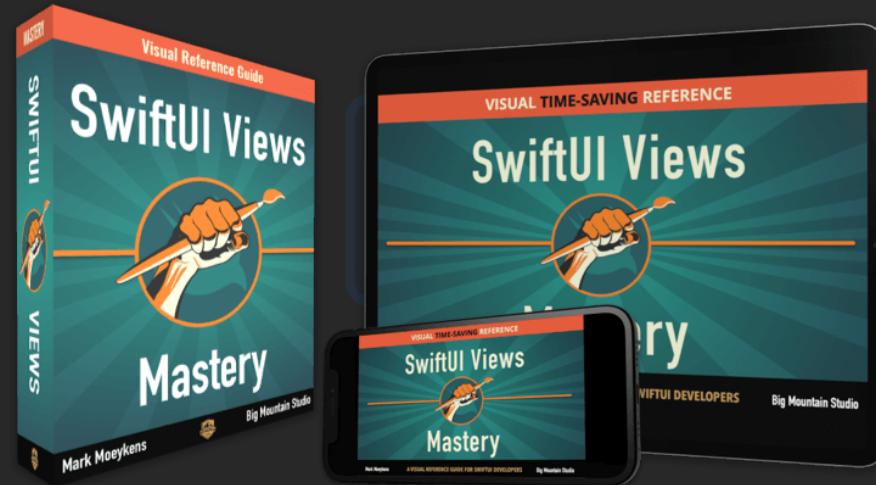
This SwiftUI content is locked in this preview.
Unlock over 20 more pages of what you can do with the Text view in the full version of the book.

Hint: If you increase the font size in the code above, it will scale LESS. That is why the first Text view scale is smaller.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**



TextEditor



This SwiftUI chapter is locked in this
preview.

You can use the TextEditor to provide text input from users that goes beyond just one line.

This is a push-out view.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

TextField



In order to get or set the text in a TextField, you need to bind it to a variable. This variable is passed into the TextField's initializer. Then, all you need to do is change this bound variable's text to change what is in the TextField. Or read the bound variable's value to see what text is currently in the TextField.

This is a push-out horizontally view.

Introduction

```
struct TextField_Intro : View {
    @State private var textFieldData = ""

    var body: some View {
        VStack(spacing: 10) {
            HeaderView("TextField", subtitle: "Introduction",
                desc: "It is required to bind text fields to a variable when using them so you can get/set the text. By default, TextFields have a plain TextStyle that has no visual content to be seen.", back: .orange)
            Image(systemName: "arrow.down.circle")
            TextField("This is a text field", text: $textFieldData)
                .padding(.horizontal)
            Image(systemName: "arrow.up.circle")

            Text("Use .textFieldStyle (.roundedBorder) to show a border.")
                .frame(maxWidth: .infinity).padding()
                .background(Color.orange)
            TextField("", text: $textFieldData)
                .textFieldStyle(.roundedBorder)
                .padding(.horizontal)
        }
        .font(.title)
    }
}
```

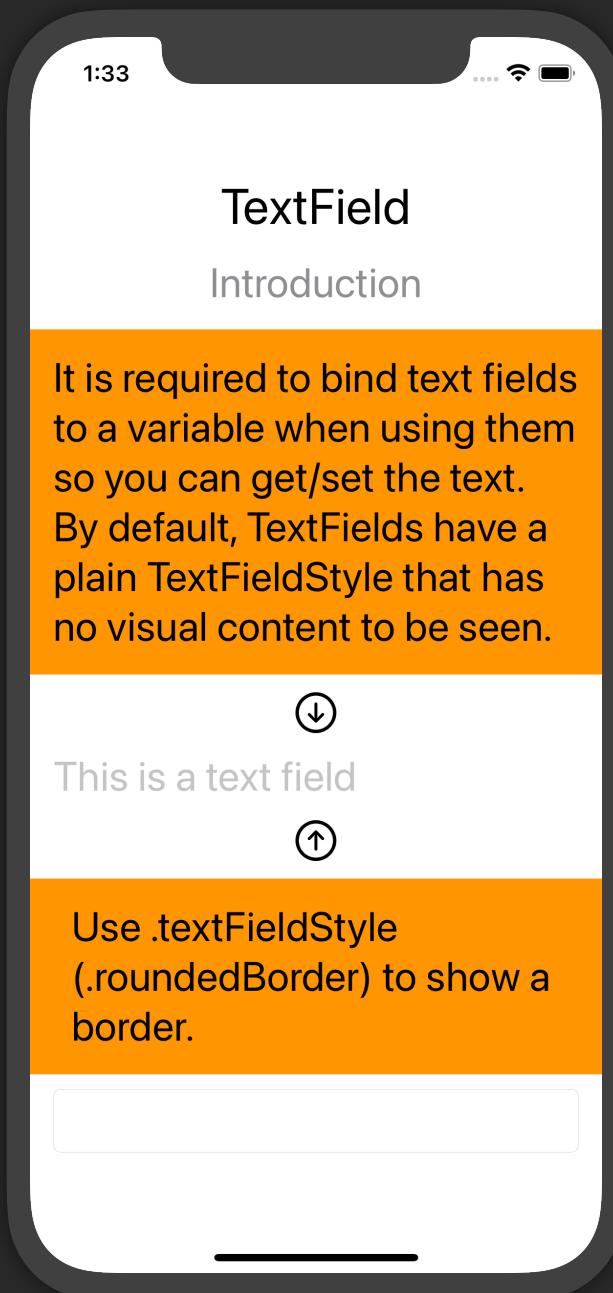
For textFieldStyle, use:

< iOS 15

RoundedBorderTextFieldStyle()

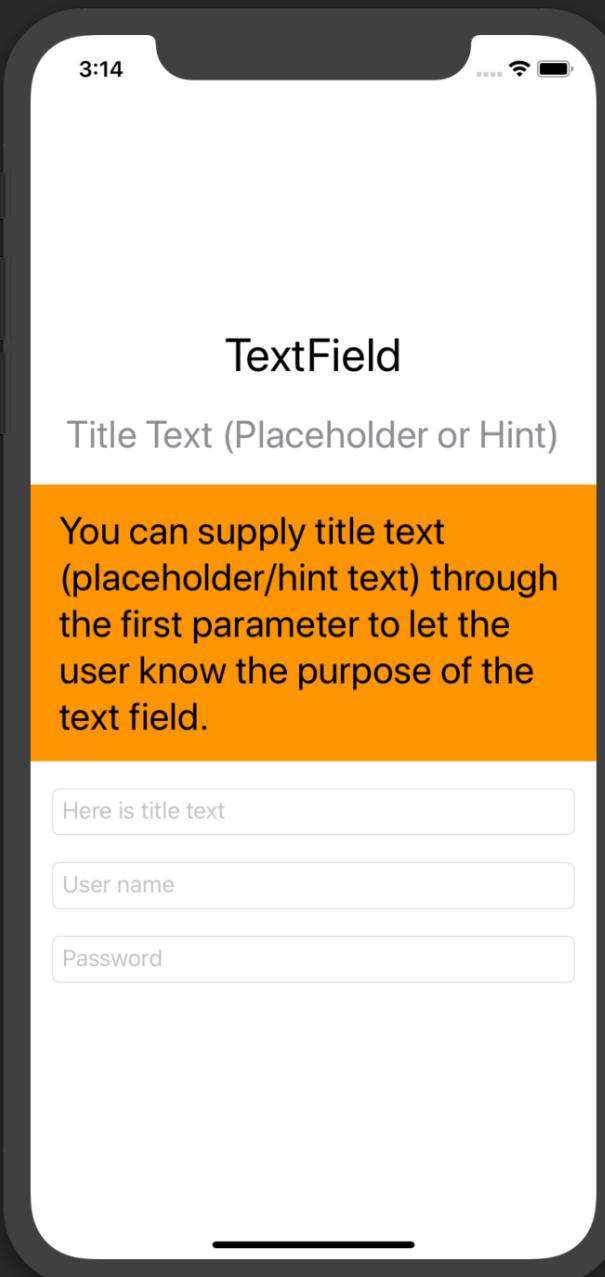
iOS 15+

.roundedBorder



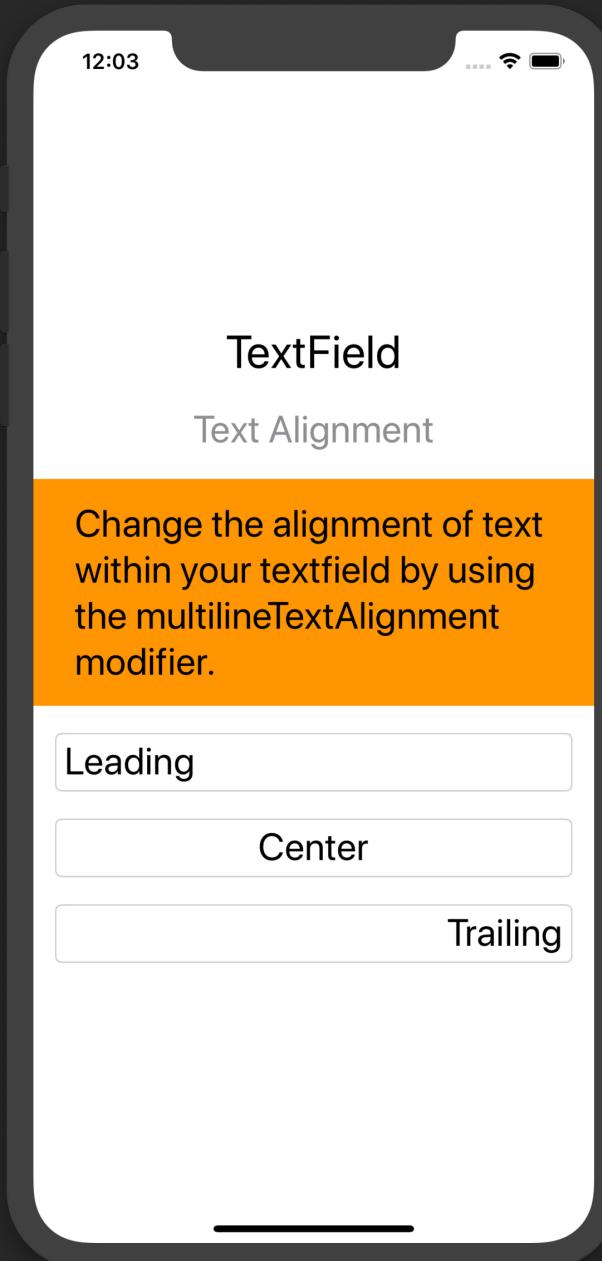
Title (Placeholder or Hint Text)

```
struct TextField_Placeholder : View {  
    @State private var textFieldData = ""  
    @State private var username = ""  
    @State private var password = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField")  
                .font(.largeTitle)  
  
            Text("Title Text (Placeholder or Hint)")  
                .foregroundColor(.gray)  
  
            Text("You can supply title text (placeholder/hint text) through the first parameter  
to let the user know the purpose of the text field.")  
                .frame(maxWidth: .infinity).padding()  
                .background(Color.orange)  
  
            Group {  
                TextField("Here is title text", text: $textFieldData)  
                    .textFieldStyle(.roundedBorder)  
  
                TextField("User name", text: $username)  
                    .textFieldStyle(.roundedBorder)  
  
                TextField("Password", text: $password)  
                    .textFieldStyle(.roundedBorder)  
            }  
            .padding(.horizontal)  
        }.font(.title)  
    }  
}
```



Text Alignment

```
struct TextField_Alignment: View {  
    @State private var textFieldData1 = "Leading"  
    @State private var textFieldData2 = "Center"  
    @State private var textFieldData3 = "Trailing"  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField").font(.largeTitle)  
            Text("Text Alignment").foregroundColor(.gray)  
            Text("Change the alignment of text within your textfield by using the  
multilineTextAlignment modifier.")  
            .frame(maxWidth: .infinity).padding()  
            .background(Color.orange)  
  
            Group {  
                TextField("Leading", text: $textFieldData1)  
                    .textFieldStyle(.roundedBorder)  
                    .multilineTextAlignment(.leading) // Default  
  
                TextField("Center", text: $textFieldData2)  
                    .textFieldStyle(.roundedBorder)  
                    .multilineTextAlignment(.center)  
  
                TextField("Trailing", text: $textFieldData3)  
                    .textFieldStyle(.roundedBorder)  
                    .multilineTextAlignment(.trailing)  
            }  
            .padding(.horizontal)  
        }.font(.title)  
    }  
}
```

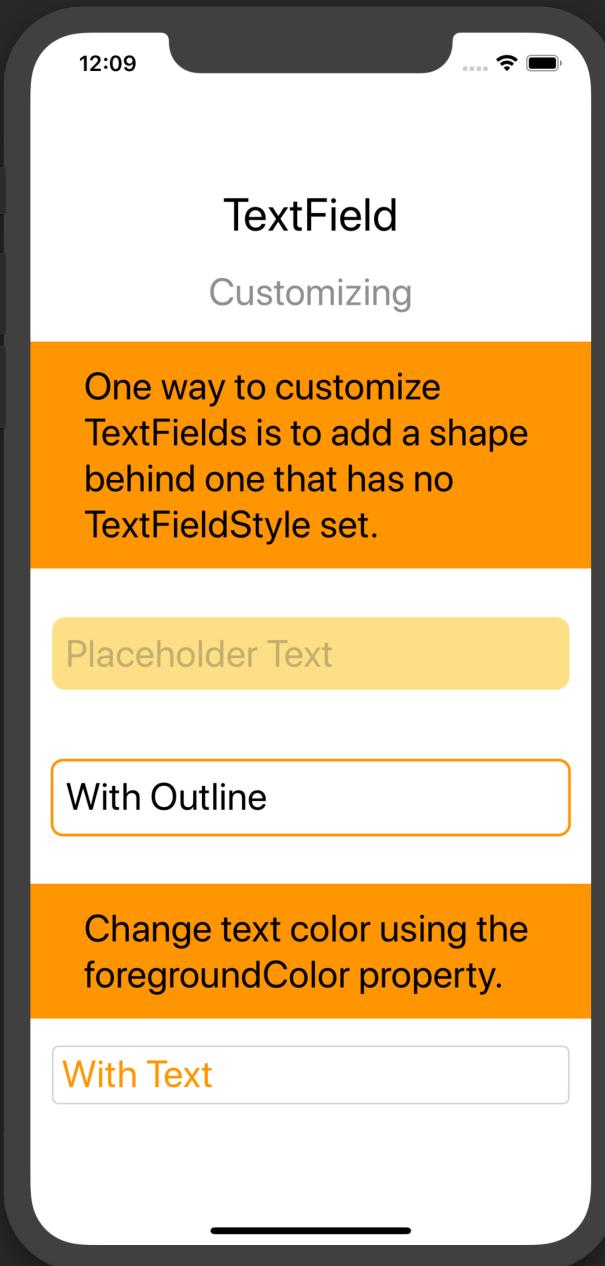




Text Size and Fonts

```
struct TextField_FontSize : View {  
    @State private var textFieldData = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField").font(.largeTitle)  
            Text("With Text Modifiers").foregroundColor(.gray)  
            Image("Font")  
            Text("To change the size of the font used within the TextField, you just need to use  
the font modifier.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.orange)  
            Group {  
                TextField("first name", text: $textFieldData)  
                    .textFieldStyle(.roundedBorder)  
  
                TextField("first name", text: $textFieldData)  
                    .font(Font.system(size: 36, design: .monospaced))  
                    .textFieldStyle(.roundedBorder)  
  
                TextField("first name", text: $textFieldData)  
                    .font(Font.system(size: 20, design: Font.Design.serif))  
                    .textFieldStyle(.roundedBorder)  
            }  
            .padding(.horizontal)  
  
            Text("Notice this also changes the placeholder or hint text in the text field.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.orange)  
            }.font(.title)  
    }  
}
```

Customizing Colors



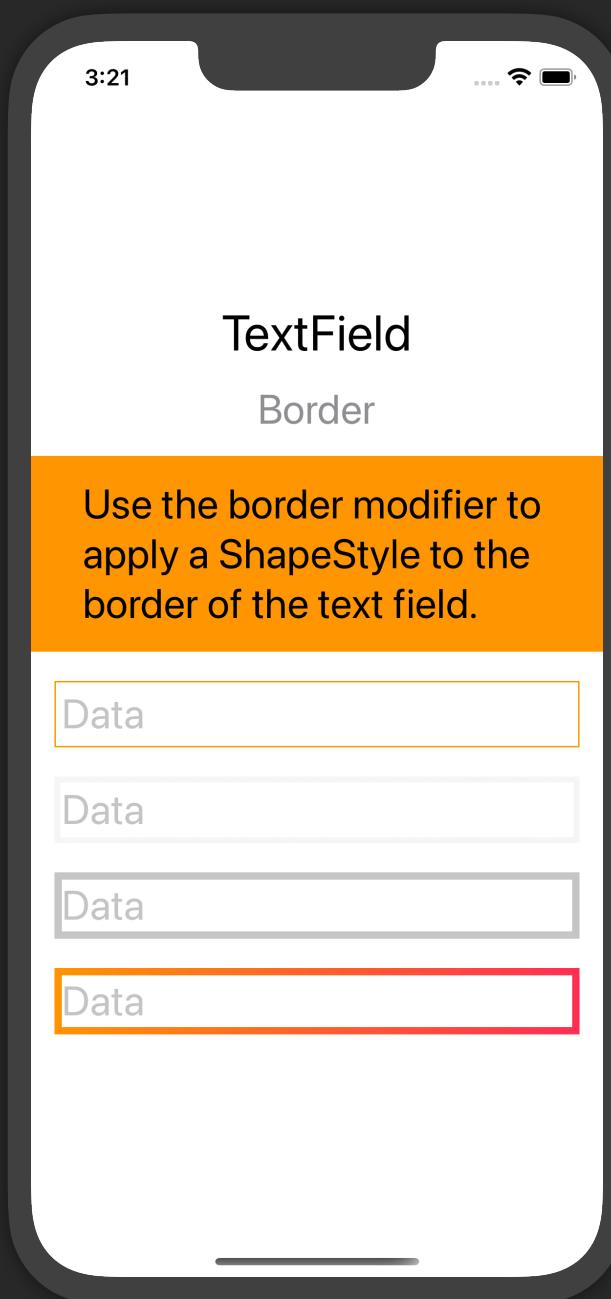
```
struct TextField_Customizing : View {  
    @State private var textFieldWithText = "With Text"  
    @State private var textFieldNoText = ""  
    @State private var withOutline = "With Outline"  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField").font(.largeTitle)  
            Text("Customizing").foregroundColor(.gray)  
            Text("One way to customize TextFields is to add a shape behind one that has no  
TextFieldStyle set.")  
                .frame(maxWidth: .infinity).padding().background(Color.orange)  
  
            TextField("Placeholder Text", text: $textFieldNoText)  
                .padding(10)  
                .background(RoundedRectangle(cornerRadius: 10)  
                    .foregroundColor(Color(hue: 0.126, saturation: 0.47, brightness: 0.993)))  
                .padding()  
            TextField("Placeholder Text", text: $withOutline)  
                .padding(10)  
                .overlay(  
                    // Add the outline  
                    RoundedRectangle(cornerRadius: 8)  
                        .stroke(Color.orange, lineWidth: 2)  
                )  
                .padding()  
  
            Text("Change text color using the foregroundColor property.")  
                .frame(maxWidth: .infinity).padding().background(Color.orange)  
  
            TextField("first name", text: $textFieldWithText)  
                .textFieldStyle(.roundedBorder)  
                .foregroundColor(.orange)  
                .padding(.horizontal)  
            }.font(.title)  
    }  
}
```

Border

```
struct TextField_Border: View {  
    @State private var textFieldData = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("TextField", subtitle: "Border",  
                      desc: "Use the border modifier to apply a ShapeStyle to the border of the  
text field.",  
                      back: .orange)  
  
            Group {  
                TextField("Data", text: $textFieldData)  
                    .padding(5)  
                    .border(.orange)  
  
                TextField("Data", text: $textFieldData)  
                    .padding(5)  
                    .border(.ultraThickMaterial, width: 4) ←  
  
                TextField("Data", text: $textFieldData)  
                    .padding(5)  
                    .border(.tertiary, width: 5)  
  
                TextField("Data", text: $textFieldData)  
                    .padding(5)  
                    .border(LinearGradient(colors: [.orange, .pink],  
                                         startPoint: .topLeading,  
                                         endPoint: .bottomTrailing), width: 5)  
            }  
            .padding(.horizontal)  
        }  
        .font(.title)  
    }  
}
```

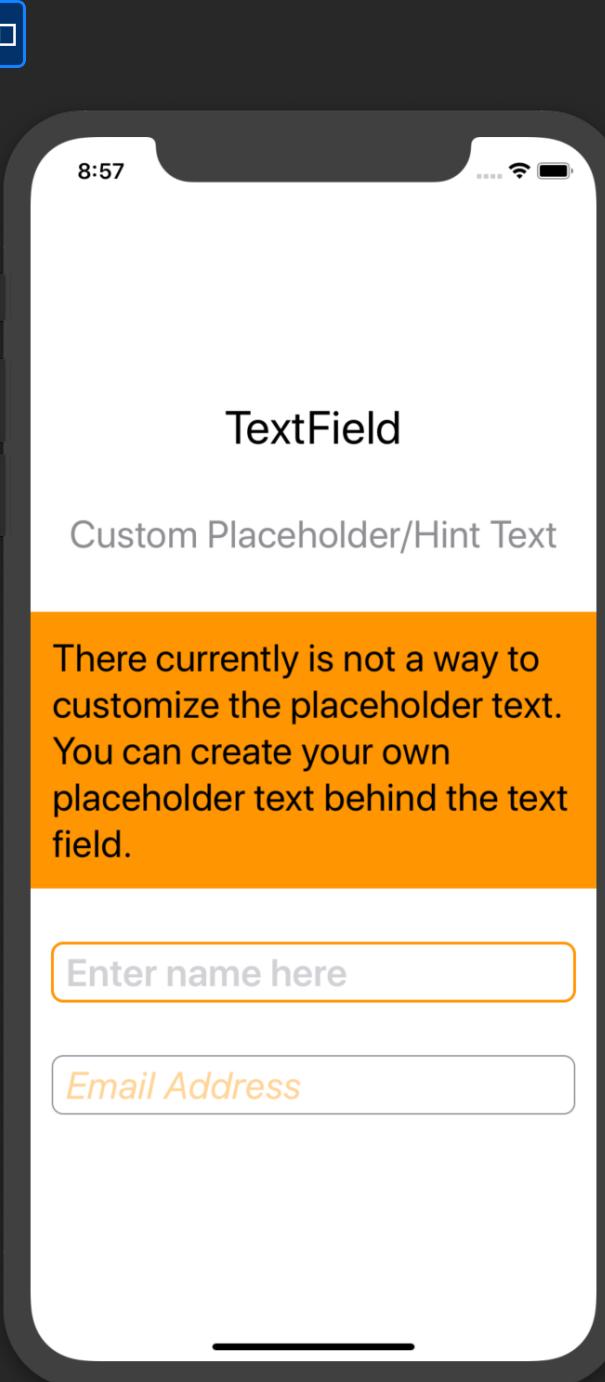
iOS 15

Material ShapeStyles
are available in iOS 15.



Custom Placeholder/Hint Text

```
struct TextField_CustomPlaceholder: View {  
    @State private var textFieldData = ""  
  
    var body: some View {  
        VStack(spacing: 40) {  
            Text("TextField").font(.largeTitle)  
            Text("Custom Placeholder/Hint Text").foregroundColor(.gray)  
            Text("There currently is not a way to customize the placeholder text. You can create  
your own placeholder text behind the text field.")  
                .frame(maxWidth: .infinity).padding().background(Color.orange)  
  
            Group {  
                // First TextField  
                ZStack(alignment: .leading) {  
                    // Only show custom hint text if there is no text entered  
                    if textFieldData.isEmpty {  
                        Text("Enter name here").bold()  
                            .foregroundColor(Color(.systemGray4))  
                    }  
                    TextField("", text: $textFieldData)  
                }  
                .padding(EdgeInsets(top: 4, leading: 10, bottom: 4, trailing: 10))  
                .overlay(  
                    // Add the outline  
                    RoundedRectangle(cornerRadius: 8)  
                        .stroke(Color.orange, lineWidth: 2))  
  
                // Second TextField  
                ZStack(alignment: .leading) {  
                    if textFieldData.isEmpty {  
                        Text("Email Address").italic()  
                    }  
                }  
            }  
        }  
    }  
}
```

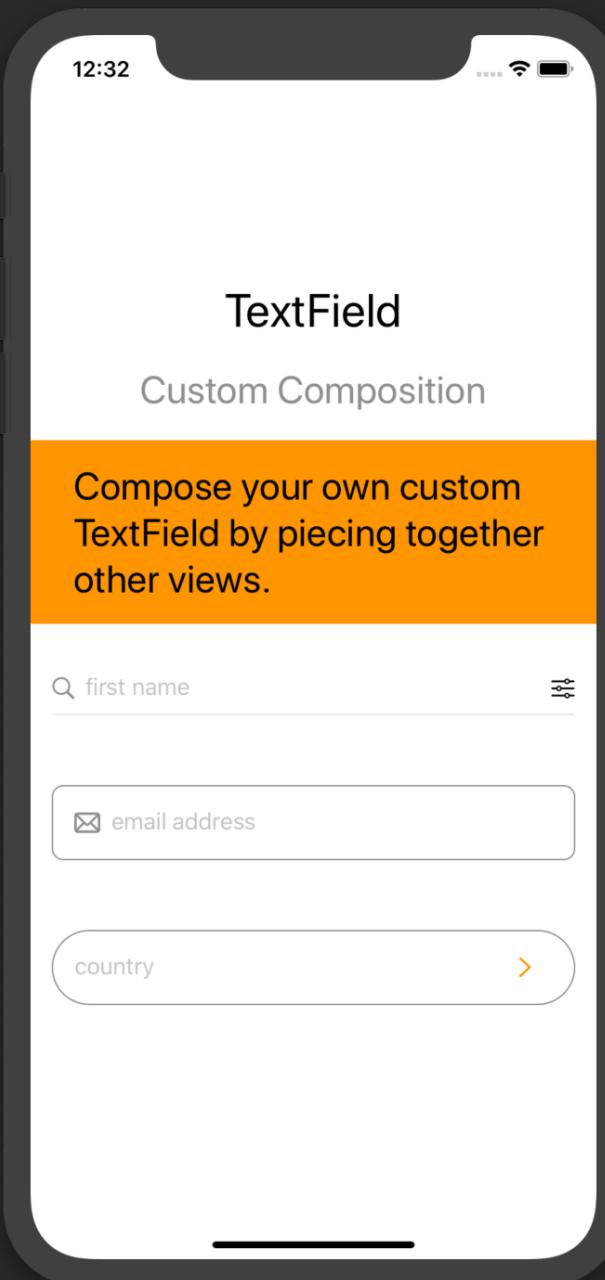




Custom Placeholder/Hint Text Continued

```
        .foregroundColor(.orange)
        .opacity(0.4)
    }
    TextField("", text: $textFieldData)
}
.padding(EdgeInsets(top: 4, leading: 10, bottom: 4, trailing: 10))
.overlay(
    RoundedRectangle(cornerRadius: 8)
        .stroke(Color.gray, lineWidth: 1))
.padding(.horizontal)
}.font(.title)
}
```

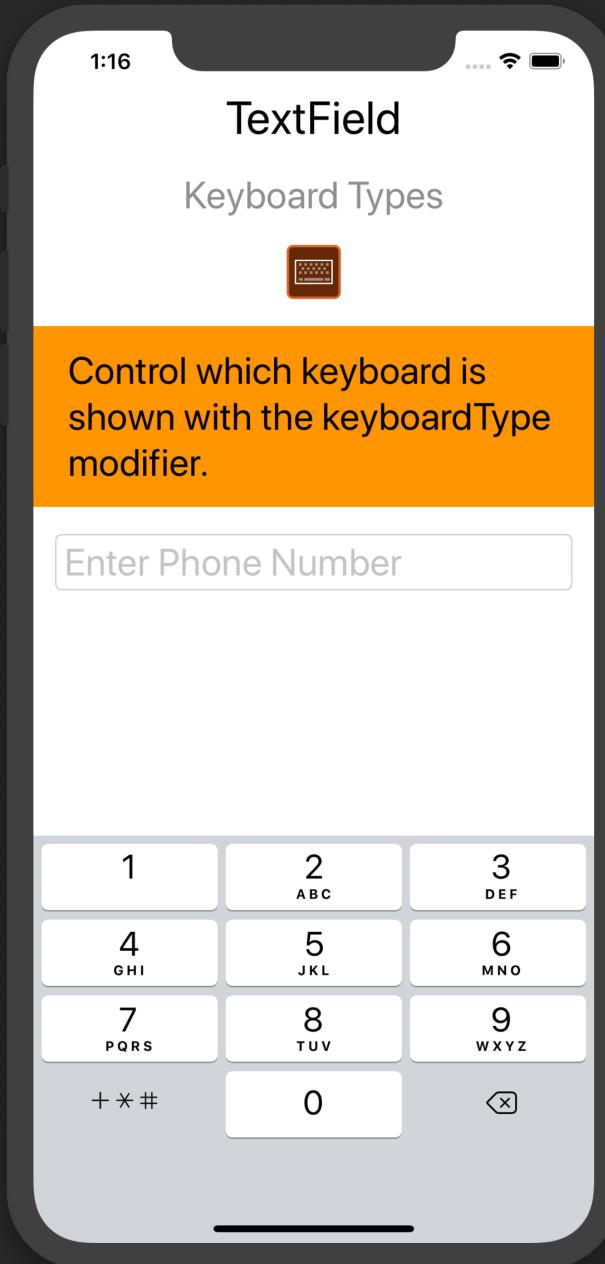
Custom Composition



```
@State private var textFieldData = ""  
...  
VStack {  
    HStack {  
        Image(systemName: "magnifyingglass").foregroundColor(.gray)  
        TextField("first name", text: $textFieldData)  
        Image(systemName: "slider.horizontal.3")  
    }  
    Divider()  
}  
.padding()  
  
HStack {  
    Image(systemName: "envelope")  
        .foregroundColor(.gray).font(.headline)  
    TextField("email address", text: $textFieldData)  
}  
.padding()  
.overlay(RoundedRectangle(cornerRadius: 8).stroke(Color.gray, lineWidth: 1))  
.padding()  
  
HStack {  
    TextField("country", text: $textFieldData)  
    Button(action: {}) {  
        Image(systemName: "chevron.right").padding(.horizontal)  
    }  
    .accentColor(.orange)  
}  
.padding()  
.overlay(Capsule().stroke(Color.gray, lineWidth: 1))  
.padding()
```



Keyboard Type

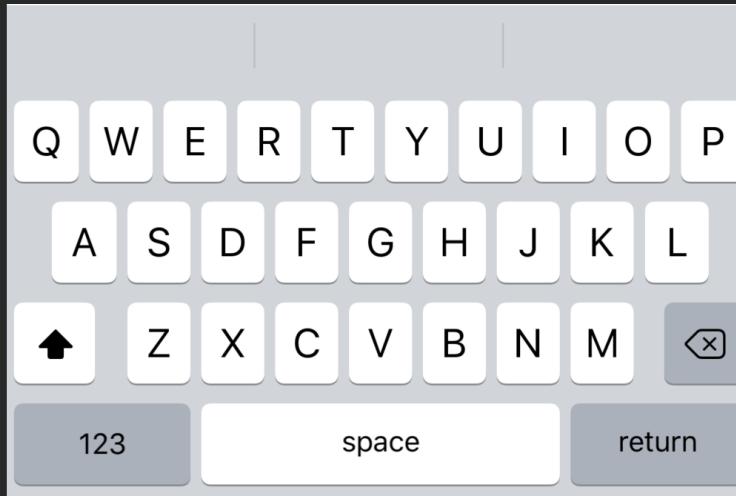


```
struct TextField_KeyboardType: View {  
    @State private var textFieldData = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField")  
                .font(.largeTitle)  
            Text("Keyboard Types")  
                .foregroundColor(.gray)  
            Image("KeyboardType")  
  
            Text("Control which keyboard is shown with the keyboardType modifier.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.orange)  
  
            TextField("Enter Phone Number", text: $textFieldData)  
                .keyboardType(UIKeyboardType.phonePad) // Show keyboard for phone numbers  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
  
            Spacer()  
        }.font(.title)  
    }  
}
```

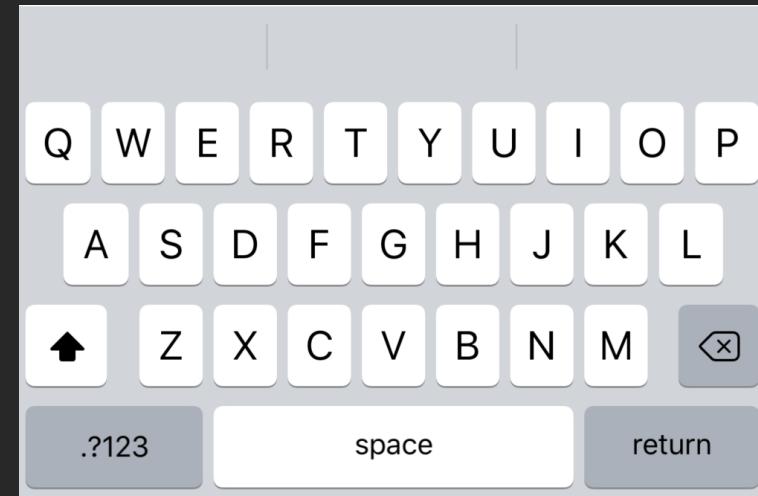


Keyboard Types Available

.default



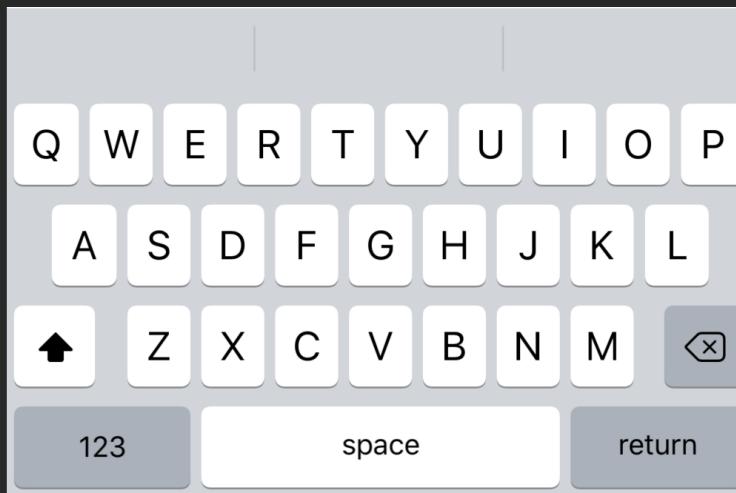
.asciiCapable



.asciiCapableNumberPad



.alphabet



.decimalPad



.emailAddress





.namePhonePad



.numberPad



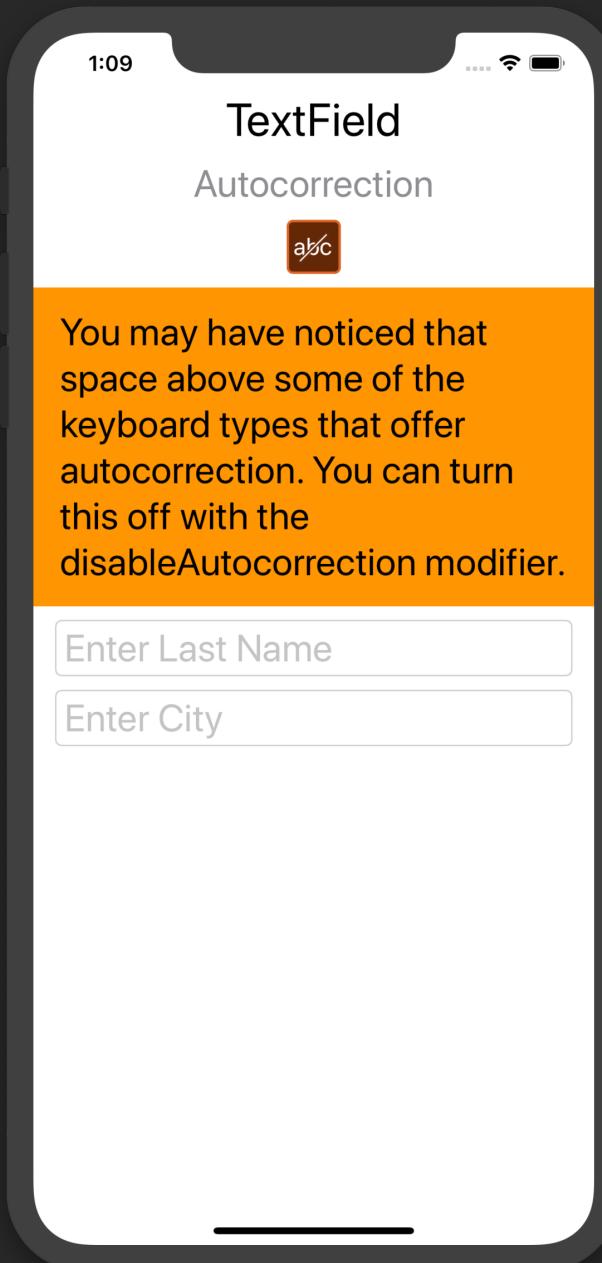
.numbersAndPunctuation



.phonePad



Disabling Autocorrect



```
struct TextField_Autocorrection: View {  
    @State private var textFieldData = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            Text("TextField")  
                .font(.largeTitle)  
            Text("Autocorrection")  
                .foregroundColor(.gray)  
            Text("You may have noticed that space above some of the keyboard types that offer autocorrection. You can turn this off with the disableAutocorrection modifier.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.orange)  
  
            TextField("Enter Last Name", text: $textFieldData)  
                .disableAutocorrection(true) // Don't offer suggestions  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
  
            TextField("Enter City", text: $textFieldData)  
                .disableAutocorrection(false) // Offer suggestions  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
  
            Spacer()  
        }.font(.title)  
    }  
}
```



TextField



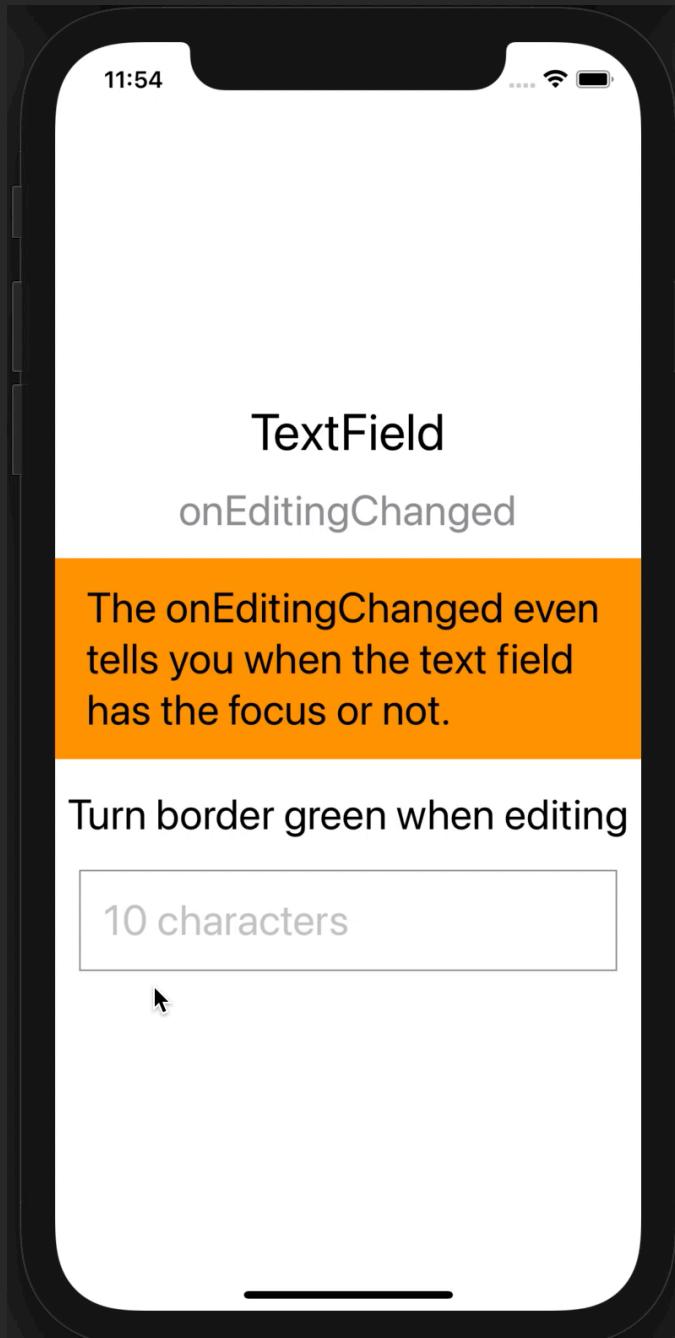
Disable TextFields

```
struct TextField_Disabled: View {  
    @State private var lastName = "Moeykens"  
    @State private var city = "Salt Lake City"  
    @State private var disabled = false  
  
    var body: some View {  
        VStack(spacing: 10) {  
            Text("TextField").font(.largeTitle)  
            Text("Disabled").foregroundColor(.gray)  
            Image("Disabled")  
            Text("You may need to conditionally enable/disable text fields. Just use the  
disabled modifier.")  
                .frame(maxWidth: .infinity)  
                .padding()  
                .background(Color.orange)  
  
            Toggle("Keep Info Private", isOn: $disabled)  
                .padding(.horizontal)  
  
            Group {  
                TextField("Enter Last Name", text: $lastName)  
                TextField("Enter City", text: $city)  
            }  
                .disableAutocorrection(true)  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
                .disabled(disabled) // Don't allow to edit when disabled  
                .opacity(disabled ? 0.5 : 1) // Fade out when disabled  
  
            Spacer()  
        }.font(.title)  
    }  
}
```

Note: The disabled modifier applies to ANY VIEW. Not just the TextField view.

onEditingChanged

```
struct TextField_OnEditingChanged: View {  
    @State private var text = ""  
    @State private var isEditing = false  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("TextField",  
                subtitle: "onEditingChanged",  
                desc: "The onEditingChanged even tells you when the text field has the  
                    focus or not.",  
                back: .orange)  
  
            Text("Turn border green when editing")  
            TextField("10 characters", text: $text) { isEditing in  
                self.isEditing = isEditing  
            }  
                .padding()  
                .border(isEditing ? Color.green : Color.gray)  
                .padding(.horizontal)  
            }  
                .font(.title)  
    }  
}
```



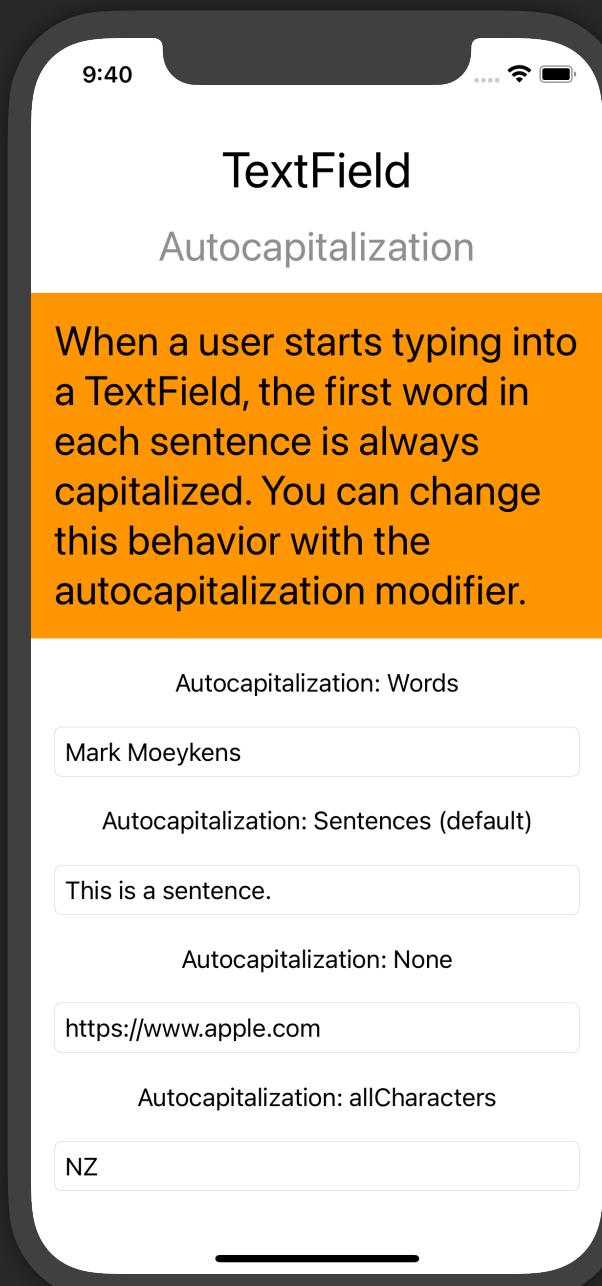
When the cursor is in the text field and the keyboard is showing, isEditing will be true.

When the keyboard is dismissed and the text field no longer has the focus, isEditing will change to false.

Autocapitalization

```
struct TextField_Autocapitalization: View {  
    @State private var textFieldData1 = ""  
    @State private var textFieldData2 = ""  
    @State private var textFieldData3 = ""  
    @State private var textFieldData4 = ""  
  
    var body: some View {  
        VStack(spacing: 20) {  
            HeaderView("TextField",  
                subtitle: "Autocapitalization",  
                desc: "When a user starts typing into a TextField, the first word in each sentence is always capitalized. You can change this behavior with the autocapitalization modifier.", back: .orange)  
                .font(.title)  
            Text("Autocapitalization: Words")  
            TextField("First & Last Name", text: $textFieldData1)  
                .autocapitalization(.words) ←  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
            Text("Autocapitalization: Sentences (default)")  
            TextField("Bio", text: $textFieldData2)  
                .autocapitalization(.sentences) ←  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
            Text("Autocapitalization: None")  
            TextField("Web Address", text: $textFieldData3)  
                .autocapitalization(.none) ←  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
            Text("Autocapitalization: allCharacters")  
            TextField("Country Code", text: $textFieldData4)  
                .autocapitalization(.allCharacters) ←  
                .textFieldStyle(.roundedBorder)  
                .padding(.horizontal)  
        }  
    }  
}
```

This is the default
autocapitalization mode.



Toggle

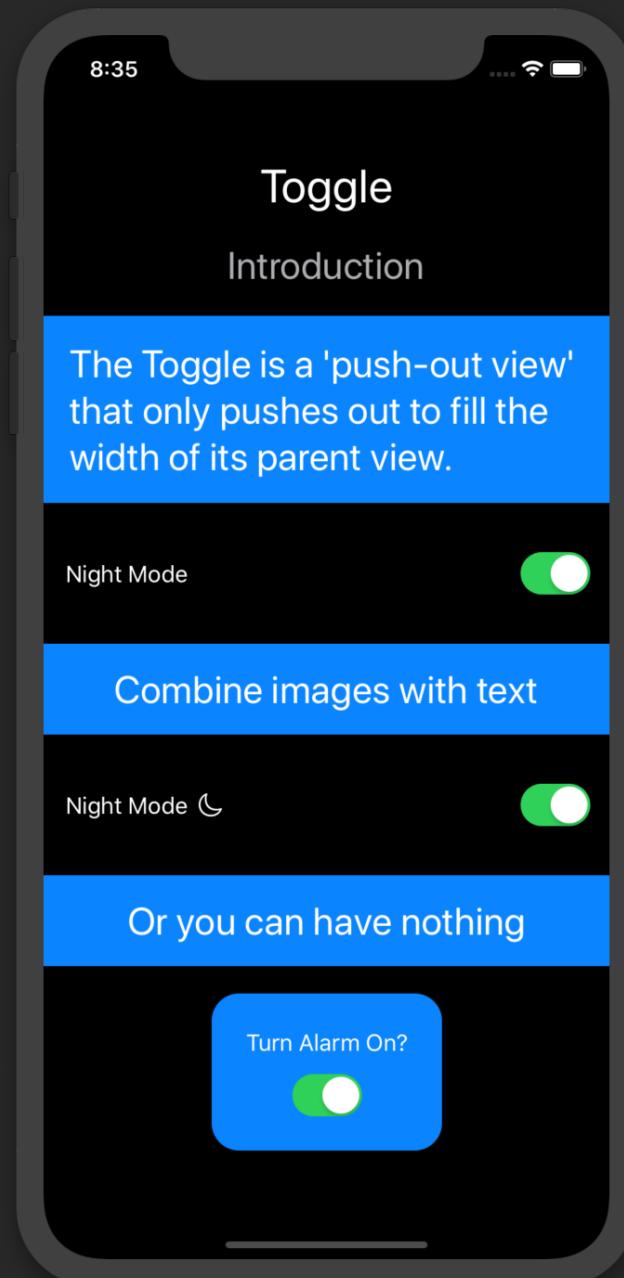


The Toggle is a switch that can either be on or off. Much like other controls, you need to bind it to a variable. This variable is passed into the Toggle's initializer. Then, all you need to do is change this bound variable's value to change the Toggle's state on or off. Or read the bound variable's value to see what state the Toggle is currently in.

This is a push-out horizontally view.

Toggle

Introduction



```
@State private var isToggleOn = true
...
Text("The Toggle fills the width of its parent view.")
...
Toggle("Night Mode", isOn: $isToggleOn)
    .padding()

Text("Combine images with text")
...
Toggle(isOn: $isToggleOn) {
    Text("Night Mode")
    Image(systemName: "moon")
}
.padding()

Text("Or you can have nothing")
...
VStack {
    Text("Turn Alarm On?")
        .foregroundColor(.white)
    Toggle("Turn this alarm on", isOn: $isToggleOn)
        .labelsHidden() // Hides the label/title
}
.padding(25)
.background(Color.blue)
.cornerRadius(20)
```

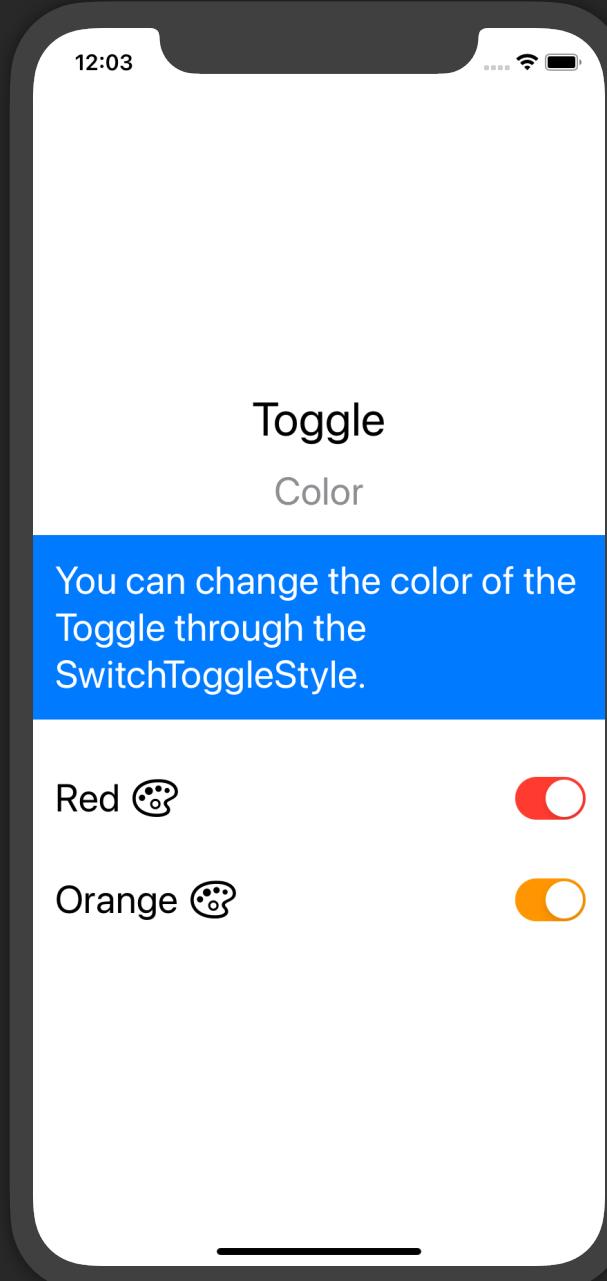
There is not a lot you can do to change the colors of the thumb (round white circle), the on and off positions. You can create your own custom Toggle. See the chapter on **Custom Styling**, in the section **ToggleStyle**.



Toggle

Accent Color

iOS 14



```
struct Toggle_Color: View {
    @State private var isToggleOn = true

    var body: some View {
        VStack(spacing: 40) {
            HeaderView("Toggle",
                      subtitle: "Color",
                      desc: "You can change the color of the Toggle through the
                             SwitchToggleStyle.", back: .blue, textColor: .white)

            Group {
                Toggle(isOn: $isToggleOn) {
                    Text("Red")
                    Image(systemName: "paintpalette")
                }
                .toggleStyle(SwitchToggleStyle(tint: Color.red))

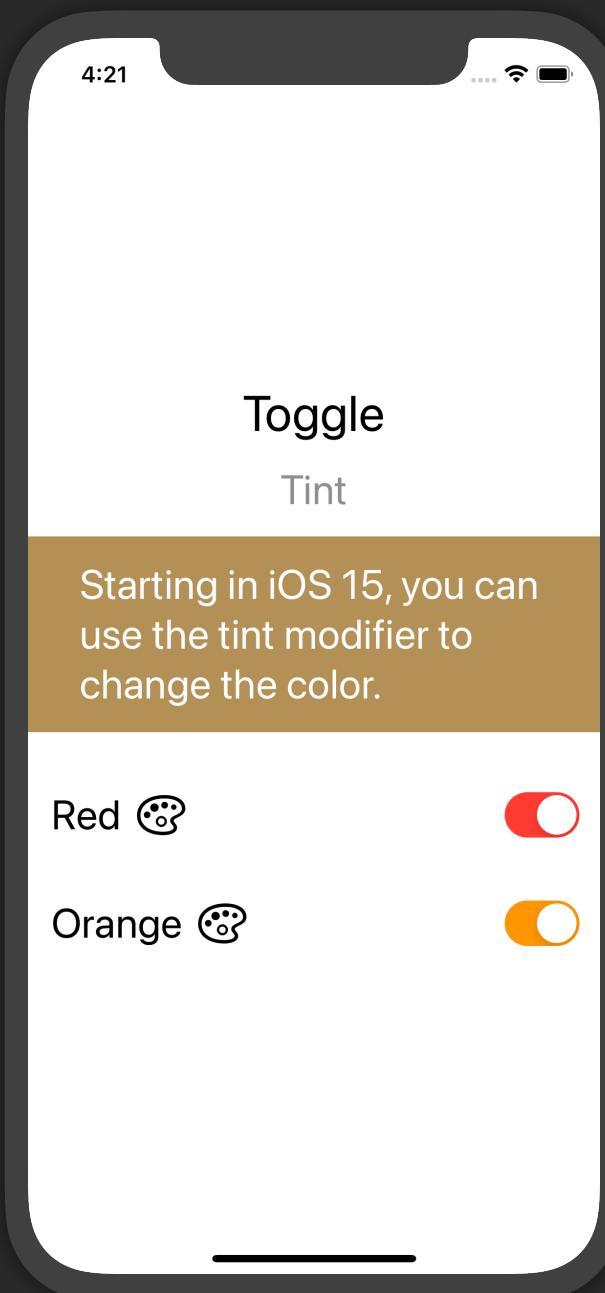
                Toggle(isOn: $isToggleOn) {
                    Text("Orange")
                    Image(systemName: "paintpalette")
                }
                .toggleStyle(SwitchToggleStyle(tint: Color.orange))
            }
            .padding(.horizontal)
        }
        .font(.title)
    }
}
```



Toggle

Tint

iOS 15



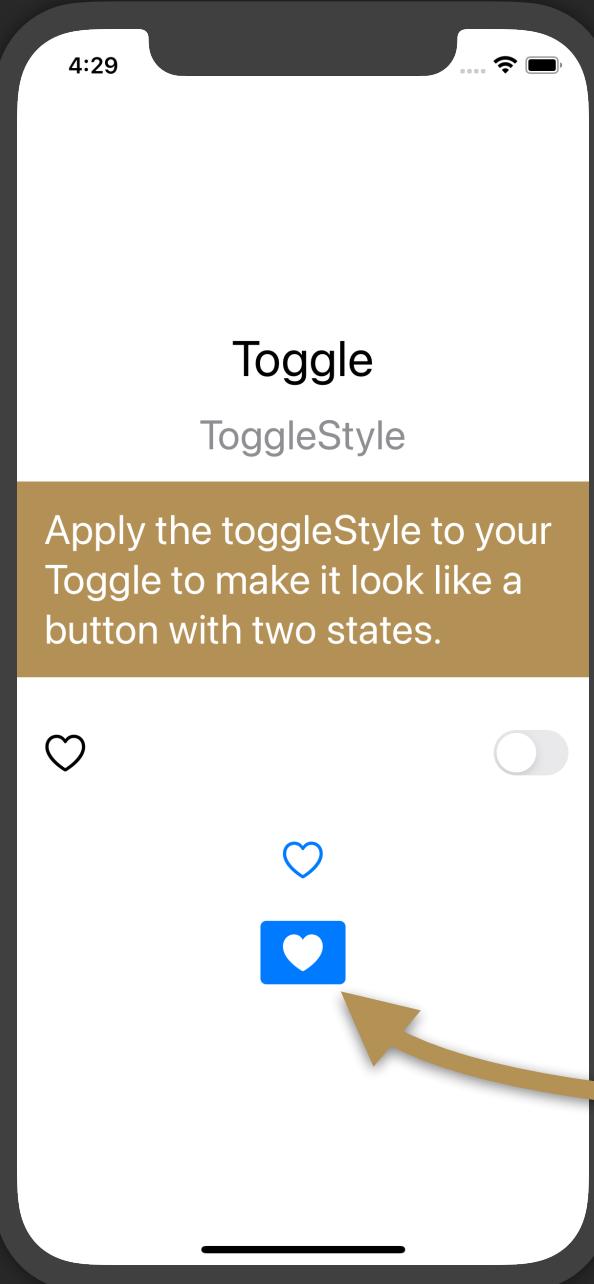
```
struct Toggle_Tint: View {  
    @State private var isToggleOn = true  
  
    var body: some View {  
        VStack(spacing: 40) {  
            HeaderView("Toggle",  
                      subtitle: "Tint",  
                      desc: "Starting in iOS 15, you can use the tint modifier to change  
                           the color.")  
  
            Group {  
                Toggle(isOn: $isToggleOn) {  
                    Text("Red")  
                    Image(systemName: "paintpalette")  
                }  
                .tint(.red)  
  
                Toggle(isOn: $isToggleOn) {  
                    Text("Orange")  
                    Image(systemName: "paintpalette")  
                }  
                .tint(.orange)  
            }  
            .padding(.horizontal)  
        }  
        .font(.title)  
    }  
}
```



Toggle

iOS 15

ToggleStyle



```
struct Toggle_ToggleStyle: View {
    @State private var isOn = false
    @State private var toggleOn = true

    var body: some View {
        VStack(spacing: 20.0) {
            HeaderView("Toggle",
                       subtitle: "ToggleStyle",
                       desc: "Apply the toggleStyle to your Toggle to make it look like a button with two states.")

            Toggle(isOn: $isOn) {
                Image(systemName: "heart")
                    .symbolVariant(isOn ? .fill : .none)
            }.padding()

            Toggle(isOn: $isOn) {
                Image(systemName: "heart")
                    .symbolVariant(isOn ? .fill : .none)
            }.toggleStyle(.button)

            Toggle(isOn: $toggleOn) {
                Image(systemName: "heart")
                    .symbolVariant(toggleOn ? .fill : .none)
            }.toggleStyle(.button)
                .font(.title)
        }
    }
}
```

These examples are using the symbol variant to switch between filled and not filled SF symbols.

Notice when the toggleStyle is button and it is in the on state, the whole button becomes filled.

ADDITIONAL CHAPTERS



The following chapters are locked in this preview.

SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

OTHER VIEWS

Covered in the **SwiftUI Views Mastery** book.

Includes: Circle, Ellipse, Capsule, Rectangle, RoundedRectangle, Color, Divider, Group, Image, Path and Inset along with the many modifiers and examples of how they work.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

PAINTS

Covered in the **SwiftUI Views Mastery** book.

Includes: AngularGradient, ImagePaint, LinearGradient and RadialGradient with the many examples of how they work when applied to different views.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

CONTROLS MODIFIER

Covered in the **SwiftUI Views Mastery** book.

Includes: ActionSheet, Alert, ContextMenu, Sheet (Modals), Popover, Custom Popups and the StatusBar
Hidden modifier with the many examples of how they work when used with different views.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

AYOUT MODIFIERS

Covered in the **SwiftUI Views Mastery** book.

Includes: AspectRatio, Background, EdgesIgnoringSafeArea, FixedSize, Frame, Hidden, LayoutPriority, Offset, Overlay, Padding, Position, ScaleToFill, ScaleToFit, and zIndex with the many examples of how they work when used with different views and modifiers.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

EFFECT MODIFIERS

Covered in the **SwiftUI Views Mastery** book.

Includes: AccentColor, BlendMode, Blur, Border, Brightness, Clipped, ClipShape, ColorInvert, ColorMultiply, ColorScheme, CompositingGroup, ContentShape, Contrast, CornerRadius, DrawingGroup, ForegroundColor, Grayscale, HueRotation, LuminanceToAlpha, Mask, Opacity, PreferredColorScheme, RotationEffect, Rotation3DEffect, Saturation, ScaleEffect, Shadow, and TransformEffect with the many examples of how they work.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

CUSTOM STYLING

Covered in the [SwiftUI Views Mastery](#) book.

Includes: ButtonStyle, DatePickerStyle, ListStyle, NavigationViewStyle, PickerStyle, TextStyle, ToggleStyle, Global Styling, View Modifiers and Styling Shapes with the many examples of how they work when used.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

IMAGE MODIFIERS

Covered in the **SwiftUI Views Mastery** book.

Includes: Interpolation, RenderingMode, Resizable, and Symbol ImageScale with the many examples of how they work.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

GESTURES

Covered in the **SwiftUI Views Mastery** book.

Includes: Drag Gesture, On Long Press Gesture, Magnification Gesture, Rotation Gesture, On Tap Gesture, Exclusive Gesture, Simultaneous Gesture, Sequence Gesture and High Priority Gesture with the examples of how they work when applied to different views.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

OTHER MODIFIERS

Covered in the **SwiftUI Views Mastery** book.

These are new modifiers that SwiftUI introduced after iOS 13. They include: AllowsHitTesting, Disabled, Preference and Redacted.

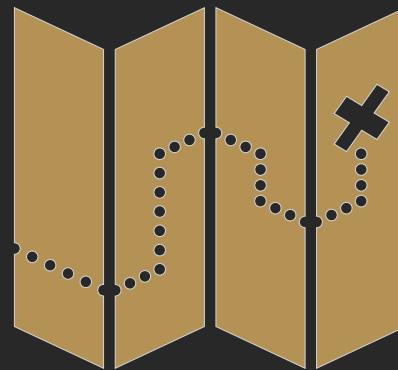
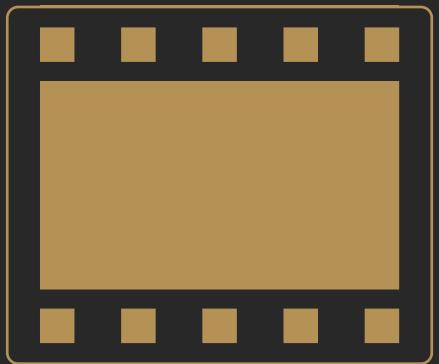


SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

IMPORTING VIEWS

Covered in the **SwiftUI Views Mastery** book.

Includes how to use the new views that came with the release of iOS 14:
VideoPlayer view in the AVKit and **Map** view in the MapKit.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

ACCESSIBILITY

Covered in the **SwiftUI Views Mastery** book.

Learn how to include accessibility to enable things like voice over and guidance to the disabled.



SAVE 10% AND UNLOCK THE BOOK TODAY FOR ONLY \$55 **\$49.50!**

THE END

I hope you enjoyed this free SwiftUI Views Quick Start!
This was just the beginning of a larger book.



Continue your journey...



Hi, I'm Mark Moeykens

I'm a full-time mobile developer with over three decades of programming experience. I have created desktop, web and mobile apps in many fields including insurance, banking, transportation, health, and sales. I have also given talks on programming and enjoy breaking down complex subjects into simple parts to teach in a practical and useful manner.



[youtube.com/markmoeykens](https://www.youtube.com/markmoeykens)

Find tutorials on iOS topics where I guide you step-by-step through all different aspects of development.



[Website: www.bigmountainstudio.com](http://www.bigmountainstudio.com)

Join my climber's camp and see what products I have available, learn something new and see what I am working on.

- *Read articles*
- *Find courses*
- *Download books*



[@BigMtnStudio](#)

Stay up-to-date on what I'm learning and working on. These are the most real-time updates you will find.



[@BigMtnStudio](#)

Do you prefer hanging out in Instagram? Then follow and get bite-sized chunks of dev info.

MORE FROM ME

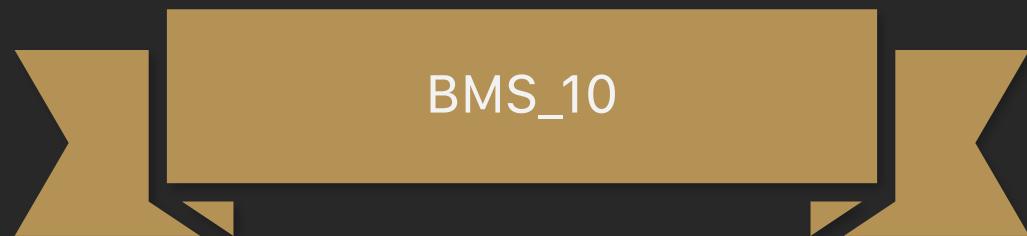


I have some products you might also be interested in!

Go to [**Big Mountain Studio**](#) to discover more.

You Get 10% Off!

Take Advantage of This Introductory Discount!



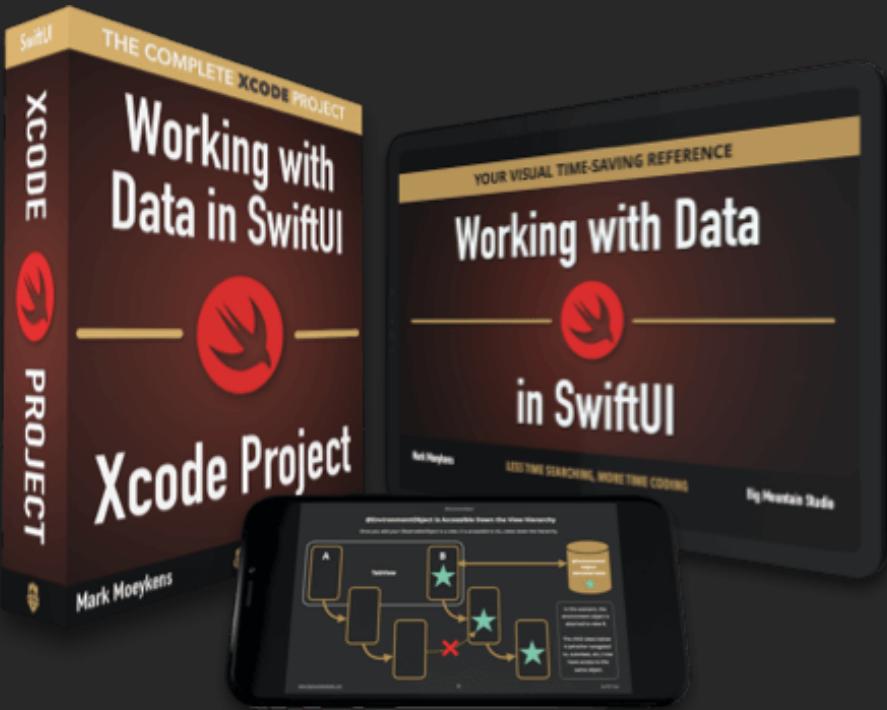
Because you got this book you get an introductory discount of **10% off everything** in the store. This is to encourage you to continue your SwiftUI journey and keep getting better at your craft. Enter the code above on checkout or click the button below. 

[ACTIVATE DISCOUNT](#)

(**Note:** You should have gotten an email with this coupon code too. Make sure you opt-in for even better discounts through sales in the future. Go [here](#) for instructions on how to do this.)

"Working with Data in SwiftUI"

DATA IN SWIFTUI DOESN'T HAVE TO BE CONFUSING. **YOU WILL MAKE MISTAKES.** BUT YOU DON'T HAVE TO WITH THIS BOOK.



Working with data in SwiftUI is super confusing. I know, I was there trying to sort it all out. That's why I made this simple to read book so that anyone can learn it.

- ✓ How to architect your app
- ✓ Learn what binding is
- ✓ What is @StateObject and when should you use it?
- ✓ How is @State different from @StateObject?
- ✓ How can you have data update automatically from parent to child views?
- ✓ How can you work with a data model and still be able to preview your views while creating them?
- ✓ How do you persist data even after your app shuts down?
- ✓ Working with JSON

[SAVE 10% ON THIS BOOK!](#)



SwiftUI Views Mastery

THE COMPLETE, VISUAL TIME-SAVING REFERENCE



- ✓ Over **1,000** pages of SwiftUI
- ✓ Over **700** screenshots/videos showing you what you can do so you can quickly come back and reference the code
- ✓ Learn all the ways to work with and modify images
- ✓ See the many ways you can use color as views
- ✓ Discover the different gradients and how you can apply them

- ✓ Find out how to implement action sheets, modals, popovers and custom popups
- ✓ Master all the layout modifiers including background and overlay layers, scaling, offsets padding and positioning
- ✓ How do you hide the status bar in SwiftUI? Find out!
- ✓ ***This is just the tip of the mountain!***

[SAVE 10% ON THIS BOOK!](#)

SwiftUI Animations Mastery

DO YOU LIKE ANIMATIONS? WOULD YOU LIKE TO SEE HUNDREDS OF VIDEO ANIMATION EXAMPLES WITH THE CODE?



SwiftUI made animations super easy...except when it isn't. Most new SwiftUI developers can get a simple animation working but as soon as they want to customize it, they get lost and waste more time than they should trying to get it working. This book will help you with that struggle.

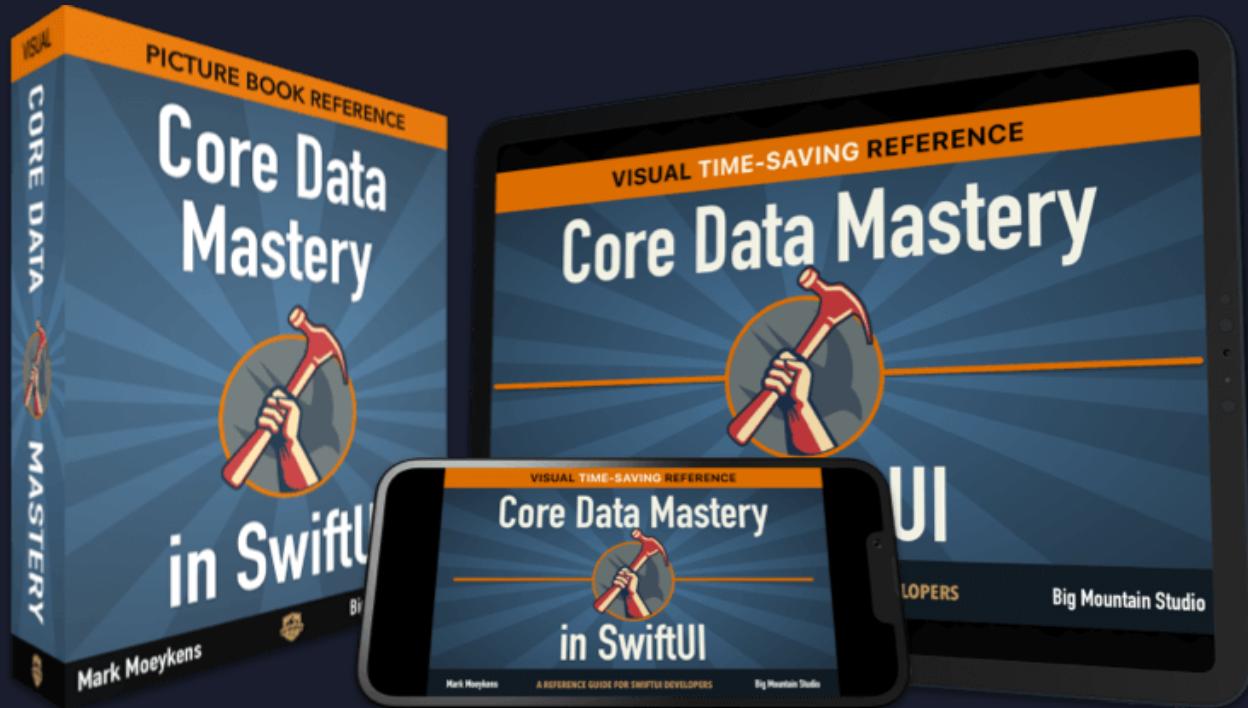
- ✓ Learn all the animation types and options with embedded video samples and code
- ✓ Master spring animations
- ✓ Master transitions for views that are inserted and removed from the screen
- ✓ Learn how matchedGeometryReader should really work
- ✓ Customize animations with speeds, delays, and durations

[SAVE 10% ON THIS BOOK!](#)



Core Data Mastery in SwiftUI

QUICKLY LEARN APPLE'S CORE DATA FRAMEWORK VISUALLY SO YOU CAN SAVE TIME BUILDING APP FEATURES.



- ✓ Over **500** pages - The largest Core Data book for SwiftUI
- ✓ What are the 4 main concepts that will help you finally understand how Core Data works?
- ✓ How can you start using Core Data in just 4 steps and under 10 minutes?
- ✓ How can you use mock data in Core Data and preview it while developing your UI?
- ✓ How can you make your life much easier when using Core Data and SwiftUI?
- ✓ How can you not only get data, but also sort, filter, and animate with a fetch request?
- ✓ What can you do if you want to group your data into sections?
- ✓ How can you use the managed object context to insert, update, and delete data?
- ✓ Much, much more. Click the link to see.

[SAVE 10% ON THIS BOOK!](#)

Combine Mastery in SwiftUI

HAVE YOU TRIED TO LEARN COMBINE AND FAILED LIKE I DID...MULTIPLE TIMES?



I finally figured out the secret to understanding Combine after 2 years and now I'm able to share it with you in this visual, game-changing reference guide.

- ✓ How can you architect your apps to work with Combine?
- ✓ Which Swift language topics should you know specifically that will allow you to understand how Combine works?
- ✓ What are the important 3 parts of Combine that allows you to build new data flows?
- ✓ How can Combine kick off multiple API calls all at one time and handle all incoming data to show on your screen? Using about **12 lines of code**...which includes error handling.

SAVE 10% ON THIS BOOK!

THANK YOU



I hope you have enjoyed this book as your visual quick start reference guide.

A lot of time and work went into this to make it as easy as possible for you to use.



If you find anything wrong or have suggestions for improvement, please let me know.

Email:
mark@bigmountainstudio.com

Direct Message me on Twitter:
[@bigmtnstudio](https://twitter.com/bigmtnstudio)



Found a way to create a cool UI? I'd be super interested to see it!

If you would like to write a positive review on how this book has helped you, I would love to hear that too!

PARTNERING INFO

IT'S FREE TO PARTNER UP AND MAKE MONEY. JUST FOLLOW THESE 2-STEPS:



1. Go to bigmountainstudio.com/makemoney and sign up.



2. Start sharing content (social posts, video mentions, blogs, etc.) with your partner link and earn passive income.

SHARE THE LOVE



If you like this book and find it useful, I encourage you to share links to my products. I'll give you some effective sharing tips on the next page. Here are some tips on what you can and cannot share.



YOU CAN

- Share a screenshot of a page from the free book sample with your followers and friends along with a link on where they can get the free or paid book. (<https://www.bigmountainstudio.com>)
- Share favorable comments and goodwill with others. You helped make the Swift international community super friendly. Let's keep it that way.



YOU CANNOT

- Share the entire book, whether it's the free sample or paid version.
- Sell or resell my products.
- Connect myself or my products with sites that promote general hate, cyber-bullying, or discrimination of any kind.

SHARING TIPS



If you have never been an affiliate before and want some tips to effectively share my products then read on!



YOU SHOULD

- Be honest and genuine with your comments. People can tell when you're not genuine.
- Be specific in your comments. Tell us exactly what you liked, what you learned or what helped you.
- Share screenshots, gifs or video of your work!
- Help others with what you learned. Add value.
- Be open that you are sharing an affiliate link.
- Share more than one time.



YOU SHOULD NOT

- Be pushy or overly aggressive.
- Make people feel wrong or stupid for not buying a product.
- Be deceptive in any way.
- Endlessly promote. Mix it in with your other content.
- Think you'll make money by sharing one time. You should casually share regularly. Try with once a week, for example.